The LaTeX3 Sources

The LaTeX Project*

Released 2024-11-02

Abstract

This is the typset sources for the expl3 programming environment; see the matching interface3 PDF for the API reference manual. The expl3 modules set up a naming scheme for LaTeX commands, which allow the LaTeX programmer to systematically name functions and variables, and specify the argument types of functions.

The TeX and ε-TeX primitives are all given a new name according to these conventions. However, in the main direct use of the primitives is not required or encouraged: the expl3 modules define an independent low-level LaTeX3 programming language.

The expl3 modules are designed to be loaded on top of LaTeX 2_ε. With an up-to-date LaTeX 2_ε kernel, this material is loaded as part of the format. The fundamental programming code can also be loaded with other TeX formats, subject to restrictions on the full range of functionality.

*E-mail: latex-team@latex-project.org

Contents

Part I

Introduction

Chapter 1

Introduction to expl3 and this document

This document is intended to act as a comprehensive reference manual for the expl3 language. A general guide to the LaTeX3 programming language is found in expl3.pdf.

1.1 Naming functions and variables

LaTeX3 does not use `@` as a "letter" for defining internal macros. Instead, the symbols `_` and `:` are used in internal macro names to provide structure. The name of each *function* is divided into logical units using `_`, while `:` separates the *name* of the function from the *argument specifier* ("arg-spec"). This describes the arguments expected by the function. In most cases, each argument is represented by a single letter. The complete list of arg-spec letters for a function is referred to as the *signature* of the function.

Each function name starts with the *module* to which it belongs. Thus apart from a small number of very basic functions, all expl3 function names contain at least one underscore to divide the module name from the descriptive name of the function. For example, all functions concerned with comma lists are in module `clist` and begin `\clist_`.

Every function must include an argument specifier. For functions which take no arguments, this will be blank and the function name will end `:`. Most functions take one or more arguments, and use the following argument specifiers:

`N` and `n` These mean *no manipulation*, of a single token for `N` and of a set of tokens given in braces for `n`. Both pass the argument through exactly as given. Usually, if you use a single token for an `n` argument, all will be well.

`c` This means *csname*, and indicates that the argument will be turned into a csname before being used. So `\foo:c {ArgumentOne}` will act in the same way as `\foo:N \ArgumentOne`. All macros that appear in the argument are expanded. An internal error will occur if the result of expansion inside a `c`-type argument is not a series of character tokens.

`V` and `v` These mean *value of variable*. The `V` and `v` specifiers are used to get the content of a variable without needing to worry about the underlying TeX structure containing the data. A `V` argument will be a single token (similar to `N`), for example

`\foo:V \MyVariable`; on the other hand, using `v` a csname is constructed first, and then the value is recovered, for example `\foo:v {MyVariable}`.

`o` This means *expansion once*. In general, the `V` and `v` specifiers are favoured over `o` for recovering stored information. However, `o` is useful for correctly processing information with delimited arguments.

`x` The `x` specifier stands for *exhaustive expansion*: every token in the argument is fully expanded until only unexpandable ones remain. The TeX `\edef` primitive carries out this type of expansion. Functions which feature an `x`-type argument are *not* expandable.

`e` The `e` specifier is in many respects identical to `x`, but uses `\expanded` primitive. Parameter character (usually `#`) in the argument need not be doubled. Functions which feature an `e`-type argument may be expandable.

`f` The `f` specifier stands for *full expansion*, and in contrast to `x` stops at the first non-expandable token (reading the argument from left to right) without trying to expand it. If this token is a ⟨*space token*⟩, it is gobbled, and thus won't be part of the resulting argument. For example, when setting a token list variable (a macro used for storage), the sequence

```
\tl_set:Nn \l_mya_tl { A }
\tl_set:Nn \l_myb_tl { B }
\tl_set:Nf \l_mya_tl { \l_mya_tl \l_myb_tl }
```

will leave `\l_mya_tl` with the content `A\l_myb_tl`, as `A` cannot be expanded and so terminates expansion before `\l_myb_tl` is considered.

`T` and `F` For logic tests, there are the branch specifiers `T` (*true*) and `F` (*false*). Both specifiers treat the input in the same way as `n` (no change), but make the logic much easier to see.

`p` The letter `p` indicates TeX *parameters*. Normally this will be used for delimited functions as expl3 provides better methods for creating simple sequential arguments.

`w` Finally, there is the `w` specifier for *weird* arguments. This covers everything else, but mainly applies to delimited values (where the argument must be terminated by some specified string).

`D` The `D` stands for **Do not use**. All of the TeX primitives are initially `\let` to a `D` name, and some are then given a second name. These functions have no standardized syntax, they are engine dependent and their name can change without warning, thus their use is *strongly discouraged* in package code: programmers should instead use the interfaces documented in interface3.pdf.

Notice that the argument specifier describes how the argument is processed prior to being passed to the underlying function. For example, `\foo:c` will take its argument, convert it to a control sequence and pass it to `\foo:N`.

Variables are named in a similar manner to functions, but begin with a single letter to define the type of variable:

`c` Constant: global parameters whose value should not be changed.

g Parameters whose value should only be set globally.

l Parameters whose value should only be set locally.

Each variable name is then build up in a similar way to that of a function, typically starting with the module[1] name and then a descriptive part. Variables end with a short identifier to show the variable type:

bitset a set of bits (a string made up of a series of `0` and `1` tokens that are accessed by position).

clist Comma separated list.

dim "Rigid" lengths.

fp Floating-point values;

int Integer-valued count register.

muskip "Rubber" lengths for use in mathematics.

skip "Rubber" lengths.

str String variables: contain character data.

tl Token list variables: placeholder for a token list.

Applying `V`-type or `v`-type expansion to variables of one of the above types is supported, while it is not supported for the following variable types:

bool Either true or false.

box Box register.

coffin A "box with handles" — a higher-level data type for carrying out `box` alignment operations.

flag Non-negative integer that can be incremented expandably.

fparray Fixed-size array of floating point values.

intarray Fixed-size array of integers.

ior/iow An input or output stream, for reading from or writing to, respectively.

prop Property list: analogue of dictionary or associative arrays in other languages.

regex Regular expression.

seq "Sequence": a data type used to implement lists (with access at both ends) and stacks.

[1]The module names are not used in case of generic scratch registers defined in the data type modules, e.g., the `int` module contains some scratch variables called `\l_tmpa_int`, `\l_tmpb_int`, and so on. In such a case adding the module name up front to denote the module and in the back to indicate the type, as in `\l_int_tmpa_int` would be very unreadable.

1.1.1 Scratch variables

Modules focussed on variable usage typically provide four scratch variables, two local and two global, with names of the form `\⟨scope⟩_tmpa_⟨type⟩`/`\⟨scope⟩_tmpb_⟨type⟩`. These are never used by the core code. The nature of TeX grouping means that as with any other scratch variable, these should only be set and used with no intervening third-party code.

There are two more special types of constants:

`q` Quark constants.

`s` Scan mark constants.

Similarly, each quark or scan mark name starts with the module name, but doesn't end with a variable type, because the type is already marked by the prefix `q` or `s`. Some general quarks and scan marks provided by LaTeX3 don't start with a module name, for example `\s_stop`. See documentation of quarks and scan marks in Chapter VII for more info.

1.1.2 Terminological inexactitude

A word of warning. In this document, and others referring to the expl3 programming modules, we often refer to "variables" and "functions" as if they were actual constructs from a real programming language. In truth, TeX is a macro processor, and functions are simply macros that may or may not take arguments and expand to their replacement text. Many of the common variables are *also* macros, and if placed into the input stream will simply expand to their definition as well — a "function" with no arguments and a "token list variable" are almost the same.[2] On the other hand, some "variables" are actually registers that must be initialised and their values set and retrieved with specific functions.

The conventions of the expl3 code are designed to clearly separate the ideas of "macros that contain data" and "macros that contain code", and a consistent wrapper is applied to all forms of "data" whether they be macros or actually registers. This means that sometimes we will use phrases like "the function returns a value", when actually we just mean "the macro expands to something". Similarly, the term "execute" might be used in place of "expand" or it might refer to the more specific case of "processing in TeX's stomach" (if you are familiar with the TeXbook parlance).

If in doubt, please ask; chances are we've been hasty in writing certain definitions and need to be told to tighten up our terminology.

1.2 Documentation conventions

This document is typeset with the experimental l3doc class; several conventions are used to help describe the features of the code. A number of conventions are used here to make the documentation clearer.

Each group of related functions is given in a box. For a function with a "user" name, this might read:

[2] TeXnically, functions with no arguments are `\long` while token list variables are not.

\ExplSyntaxOn
\ExplSyntaxOff

`\ExplSyntaxOn ... \ExplSyntaxOff`

The textual description of how the function works would appear here. The syntax of the function is shown in mono-spaced text to the right of the box. In this example, the function takes no arguments and so the name of the function is simply reprinted.

For programming functions, which use `_` and `:` in their name there are a few additional conventions: If two related functions are given with identical names but different argument specifiers, these are termed *variants* of each other, and the latter functions are printed in grey to show this more clearly. They will carry out the same function but will take different types of argument:

\seq_new:N
\seq_new:c

`\seq_new:N` ⟨*sequence*⟩

When a number of variants are described, the arguments are usually illustrated only for the base function. Here, ⟨*sequence*⟩ indicates that `\seq_new:N` expects the name of a sequence. From the argument specifier, `\seq_new:c` also expects a sequence name, but as a name rather than as a control sequence. Each argument given in the illustration should be described in the following text.

Fully expandable functions Some functions are fully expandable, which allows them to be used within an `x`-type or `e`-type argument (in plain TeX terms, inside an `\edef` or `\expanded`), as well as within an `f`-type argument. These fully expandable functions are indicated in the documentation by a star:

\cs_to_str:N ★

`\cs_to_str:N` ⟨*cs*⟩

As with other functions, some text should follow which explains how the function works. Usually, only the star will indicate that the function is expandable. In this case, the function expects a ⟨*cs*⟩, shorthand for a ⟨*control sequence*⟩.

Restricted expandable functions A few functions are fully expandable but cannot be fully expanded within an `f`-type argument. In this case a hollow star is used to indicate this:

\seq_map_function:NN ☆

`\seq_map_function:NN` ⟨*seq*⟩ ⟨*function*⟩

Conditional functions Conditional (`if`) functions are normally defined in three variants, with `T`, `F` and `TF` argument specifiers. This allows them to be used for different "true"/"false" branches, depending on which outcome the conditional is being used to test. To indicate this without repetition, this information is given in a shortened form:

\sys_if_engine_xetex:*TF* ⋆ \sys_if_engine_xetex:TF {⟨*true code*⟩} {⟨*false code*⟩}

The underlining and italic of TF indicates that three functions are available:

- \sys_if_engine_xetex:T
- \sys_if_engine_xetex:F
- \sys_if_engine_xetex:TF

Usually, the illustration will use the TF variant, and so both ⟨*true code*⟩ and ⟨*false code*⟩ will be shown. The two variant forms T and F take only ⟨*true code*⟩ and ⟨*false code*⟩, respectively. Here, the star also shows that this function is expandable. With some minor exceptions, *all* conditional functions in the expl3 modules should be defined in this way.

Variables, constants and so on are described in a similar manner:

\l_tmpa_tl A short piece of text will describe the variable: there is no syntax illustration in this case.

In some cases, the function is similar to one in LaTeX 2ε or plain TeX. In these cases, the text will include an extra "**TeXhackers note**" section:

\token_to_str:N ⋆ \token_to_str:N ⟨*token*⟩

The normal description text.

TeXhackers note: Detail for the experienced TeX or LaTeX 2ε programmer. In this case, it would point out that this function is the TeX primitive \string.

Changes to behaviour When new functions are added to expl3, the date of first inclusion is given in the documentation. Where the documented behaviour of a function changes after it is first introduced, the date of the update will also be given. This means that the programmer can be sure that any release of expl3 after the date given will contain the function of interest with expected behaviour as described. Note that changes to code internals, including bug fixes, are not recorded in this way *unless* they impact on the expected behaviour.

1.3 Formal language conventions which apply generally

As this is a formal reference guide for LaTeX3 programming, the descriptions of functions are intended to be reasonably "complete". However, there is also a need to avoid repetition. Formal ideas which apply to general classes of function are therefore summarised here.

For tests which have a TF argument specification, the test if evaluated to give a logically TRUE or FALSE result. Depending on this result, either the ⟨*true code*⟩ or the ⟨*false code*⟩ will be left in the input stream. In the case where the test is expandable, and a predicate (_p) variant is available, the logical value determined by the test is left in the input stream: this will typically be part of a larger logical construct.

1.4 TeX concepts not supported by LaTeX3

The TeX concept of an "`\outer`" macro is *not supported* at all by LaTeX3. As such, the functions provided here may break when used on top of LaTeX 2ε if `\outer` tokens are used in the arguments.

Part II
Bootstrapping

Chapter 2

The l3bootstrap module Bootstrap code

2.1 Using the LaTeX3 modules

The modules documented in `interface3` (and this file) are designed to be used on top of LaTeX 2ε and are already pre-loaded since LaTeX 2ε 2020-02-02. To support older formats, the `\usepackage{expl3}` or `\RequirePackage{expl3}` instructions are still available to load them all as one.

As the modules use a coding syntax different from standard LaTeX 2ε it provides a few functions for setting it up.

`\ExplSyntaxOn`
`\ExplSyntaxOff`

Updated: 2011-08-13

`\ExplSyntaxOn ⟨code⟩ \ExplSyntaxOff`

The `\ExplSyntaxOn` function switches to a category code regime in which spaces and new lines are ignored, and in which the colon (`:`) and underscore (`_`) are treated as "letters", thus allowing access to the names of code functions and variables. Within this environment, `~` is used to input a space. The `\ExplSyntaxOff` reverts to the document category code regime.

TeXhackers note: Spaces introduced by `~` behave much in the same way as normal space characters in the standard category code regime: they are ignored after a control word or at the start of a line, and multiple consecutive `~` are equivalent to a single one. However, `~` is *not* ignored at the end of a line.

`\ProvidesExplPackage`
`\ProvidesExplClass`
`\ProvidesExplFile`

Updated: 2023-08-03

`\ProvidesExplPackage {⟨package⟩} {⟨date⟩} {⟨version⟩} {⟨description⟩}`

These functions act broadly in the same way as the corresponding LaTeX 2ε kernel functions `\ProvidesPackage`, `\ProvidesClass` and `\ProvidesFile`. However, they also implicitly switch `\ExplSyntaxOn` for the remainder of the code with the file. At the end of the file, `\ExplSyntaxOff` will be called to reverse this. (This is the same concept as LaTeX 2ε provides in turning on `\makeatletter` within package and class code.) The ⟨*date*⟩ should be given in the format ⟨*year*⟩/⟨*month*⟩/⟨*day*⟩ or in the ISO date format ⟨*year*⟩-⟨*month*⟩-⟨*day*⟩. If the ⟨*version*⟩ is given then a leading `v` is optional: if given as a "pure" version string, a `v` will be prepended.

`\GetIdInfo`

Updated: 2012-06-04

`\GetIdInfo $Id:` ⟨*SVN info field*⟩ `$ {`⟨*description*⟩`}`

Extracts all information from a SVN field. Spaces are not ignored in these fields. The information pieces are stored in separate control sequences with `\ExplFileName` for the part of the file name leading up to the period, `\ExplFileDate` for date, `\ExplFileVersion` for version and `\ExplFileDescription` for the description.

To summarize: Every single package using this syntax should identify itself using one of the above methods. Special care is taken so that every package or class file loaded with `\RequirePackage` or similar are loaded with usual LaTeX 2_ε category codes and the LaTeX3 category code scheme is reloaded when needed afterwards. See implementation for details. If you use the `\GetIdInfo` command you can use the information when loading a package with

```
\ProvidesExplPackage{\ExplFileName}
  {\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
```

Chapter 3

The l3names module
Namespace for primitives

3.1 Setting up the LaTeX3 programming language

This module is at the core of the LaTeX3 programming language. It performs the following tasks:

- defines new names for all TeX primitives;
- emulate required primitives not provided by default in LuaTeX;
- switches to the category code régime for programming;

This module is entirely dedicated to primitives (and emulations of these), which should not be used directly within LaTeX3 code (outside of "kernel-level" code). As such, the primitives are not documented here: *The TeXbook*, *TeX by Topic* and the manuals for pdfTeX, XeTeX, LuaTeX, pTeX and upTeX should be consulted for details of the primitives. These are named `\tex_`⟨*name*⟩`:D`, typically based on the primitive's ⟨*name*⟩ in pdfTeX and omitting a leading `pdf` when the primitive is not related to pdf output.

Part III
Programming Flow

Chapter 4

The l3basics module Basic definitions

As the name suggests, this module holds some basic definitions which are needed by most or all other modules in this set.

Here we describe those functions that are used all over the place. By that, we mean functions dealing with the construction and testing of control sequences. Furthermore the basic parts of conditional processing are covered; conditional processing dealing with specific data types is described in the modules specific for the respective data types.

4.1 No operation functions

`\prg_do_nothing:` ⋆

`\prg_do_nothing:`

An expandable function which does nothing at all: leaves nothing in the input stream after a single expansion.

`\scan_stop:`

`\scan_stop:`

A non-expandable function which does nothing. Does not vanish on expansion but produces no typeset output.

4.2 Grouping material

`\group_begin:`
`\group_end:`

`\group_begin:`
`\group_end:`

These functions begin and end a group for definition purposes. Assignments are local to groups unless carried out in a global manner. (A small number of exceptions to this rule will be noted as necessary elsewhere in this document.) Each `\group_begin:` must be matched by a `\group_end:`, although this does not have to occur within the same function. Indeed, it is often necessary to start a group within one function and finish it within another, for example when seeking to use non-standard category codes.

TEXhackers note: These are the TEX primitives `\begingroup` and `\endgroup`.

`\group_insert_after:N` `\group_insert_after:N` ⟨*token*⟩

Adds ⟨*token*⟩ to the list of ⟨*tokens*⟩ to be inserted when the current group level ends. The list of ⟨*tokens*⟩ to be inserted is empty at the beginning of a group: multiple applications of `\group_insert_after:N` may be used to build the inserted list one ⟨*token*⟩ at a time. The current group level may be closed by a `\group_end:` function or by a token with category code 2 (close-group), namely a `}` if standard category codes apply.

> **TEXhackers note:** This is the TEX primitive `\aftergroup`.

`\group_show_list:` `\group_show_list:`
`\group_log_list:` `\group_log_list:`

New: 2021-05-11

Display (to the terminal or log file) a list of the groups that are currently opened. This is intended for tracking down problems.

> **TEXhackers note:** This is a wrapper around the ε-TEX primitive `\showgroups`.

4.3 Control sequences and functions

As TEX is a macro language, creating new functions means creating macros. At point of use, a function is replaced by the replacement text ("code") in which each parameter in the code (`#1`, `#2`, *etc.*) is replaced the appropriate arguments absorbed by the function. In the following, ⟨*code*⟩ is therefore used as a shorthand for "replacement text".

Functions which are not "protected" are fully expanded inside an `e`-type or `x`-type expansion. In contrast, "protected" functions are not expanded within `e` and `x` expansions.

4.3.1 Defining functions

Functions can be created with no requirement that they are declared first (in contrast to variables, which must always be declared). Declaring a function before setting up the code means that the name chosen is checked and an error raised if it is already in use. The name of a function can be checked at the point of definition using the `\cs_new...` functions: this is recommended for all functions which are defined for the first time.

There are three ways to define new functions. All classes define a function to expand to the substitution text. Within the substitution text the actual parameters are substituted for the formal parameters (`#1`, `#2`, ...).

`new` Create a new function with the `new` scope, such as `\cs_new:Npn`. The definition is global and results in an error if it is already defined.

`set` Create a new function with the `set` scope, such as `\cs_set:Npn`. The definition is restricted to the current TEX group and does not result in an error if the function is already defined.

`gset` Create a new function with the `gset` scope, such as `\cs_gset:Npn`. The definition is global and does not result in an error if the function is already defined.

Within each set of scope there are different ways to define a function. The differences depend on restrictions on the actual parameters and the expandability of the resulting function.

nopar Create a new function with the nopar restriction, such as \cs_set_nopar:Npn. The parameter may not contain \par tokens.

protected Create a new function with the protected restriction, such as \cs_set_-protected:Npn. The parameter may contain \par tokens but the function will not expand within an e-type or x-type expansion.

Finally, the functions in Subsections 4.3.2 and 4.3.3 are primarily meant to define *base functions* only. Base functions can only have the following argument specifiers:

N and n No manipulation.

T and F Functionally equivalent to n (you are actually encouraged to use the family of \prg_new_conditional: functions described in Section 9.1).

p and w These are special cases.

The \cs_new: functions below (and friends) do not stop you from using other argument specifiers in your function names, but they do not handle expansion for you. You should define the base function and then use \cs_generate_variant:Nn to generate custom variants as described in Section 5.2.

4.3.2 Defining new functions using parameter text

\cs_new:Npn
\cs_new:cpn
\cs_new:Npe
\cs_new:cpe
\cs_new:Npx
\cs_new:cpx

\cs_new:Npn ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

Creates ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed by the function. The definition is global and an error results if the ⟨*function*⟩ is already defined.

\cs_new_nopar:Npn
\cs_new_nopar:cpn
\cs_new_nopar:Npe
\cs_new_nopar:cpe
\cs_new_nopar:Npx
\cs_new_nopar:cpx

\cs_new_nopar:Npn ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

Creates ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain \par tokens. The definition is global and an error results if the ⟨*function*⟩ is already defined.

\cs_new_protected:Npn
\cs_new_protected:cpn
\cs_new_protected:Npe
\cs_new_protected:cpe
\cs_new_protected:Npx
\cs_new_protected:cpx

\cs_new_protected:Npn ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

Creates ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed by the function. The ⟨*function*⟩ will not expand within an e-type or or x-type argument. The definition is global and an error results if the ⟨*function*⟩ is already defined.

\cs_new_protected_nopar:Npn
\cs_new_protected_nopar:cpn
\cs_new_protected_nopar:Npe
\cs_new_protected_nopar:cpe
\cs_new_protected_nopar:Npx
\cs_new_protected_nopar:cpx

\cs_new_protected_nopar:Npn ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

Creates ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain \par tokens. The ⟨*function*⟩ will not expand within an e-type or x-type argument. The definition is global and an error results if the ⟨*function*⟩ is already defined.

\cs_set:Npn
\cs_set:cpn
\cs_set:Npe
\cs_set:cpe
\cs_set:Npx
\cs_set:cpx

\cs_set:Npn ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the ⟨*function*⟩ is restricted to the current TeX group level.

\cs_set_nopar:Npn
\cs_set_nopar:cpn
\cs_set_nopar:Npe
\cs_set_nopar:cpe
\cs_set_nopar:Npx
\cs_set_nopar:cpx

\cs_set_nopar:Npn ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain \par tokens. The assignment of a meaning to the ⟨*function*⟩ is restricted to the current TeX group level.

\cs_set_protected:Npn
\cs_set_protected:cpn
\cs_set_protected:Npe
\cs_set_protected:cpe
\cs_set_protected:Npx
\cs_set_protected:cpx

\cs_set_protected:Npn ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the ⟨*function*⟩ is restricted to the current TeX group level. The ⟨*function*⟩ will not expand within an e-type or x-type argument.

\cs_set_protected_nopar:Npn
\cs_set_protected_nopar:cpn
\cs_set_protected_nopar:Npe
\cs_set_protected_nopar:cpe
\cs_set_protected_nopar:Npx
\cs_set_protected_nopar:cpx

\cs_set_protected_nopar:Npn ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain \par tokens. The assignment of a meaning to the ⟨*function*⟩ is restricted to the current TeX group level. The ⟨*function*⟩ will not expand within an e-type or x-type argument.

\cs_gset:Npn
\cs_gset:cpn
\cs_gset:Npe
\cs_gset:cpe
\cs_gset:Npx
\cs_gset:cpx

\cs_gset:Npn ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

Globally sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the ⟨*function*⟩ is *not* restricted to the current TEX group level: the assignment is global.

\cs_gset_nopar:Npn
\cs_gset_nopar:cpn
\cs_gset_nopar:Npe
\cs_gset_nopar:cpe
\cs_gset_nopar:Npx
\cs_gset_nopar:cpx

\cs_gset_nopar:Npn ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

Globally sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain \par tokens. The assignment of a meaning to the ⟨*function*⟩ is *not* restricted to the current TEX group level: the assignment is global.

\cs_gset_protected:Npn
\cs_gset_protected:cpn
\cs_gset_protected:Npe
\cs_gset_protected:cpe
\cs_gset_protected:Npx
\cs_gset_protected:cpx

\cs_gset_protected:Npn ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

Globally sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the ⟨*function*⟩ is *not* restricted to the current TEX group level: the assignment is global. The ⟨*function*⟩ will not expand within an e-type or x-type argument.

\cs_gset_protected_nopar:Npn
\cs_gset_protected_nopar:cpn
\cs_gset_protected_nopar:Npe
\cs_gset_protected_nopar:cpe
\cs_gset_protected_nopar:Npx
\cs_gset_protected_nopar:cpx

\cs_gset_protected_nopar:Npn ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

Globally sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain \par tokens. The assignment of a meaning to the ⟨*function*⟩ is *not* restricted to the current TEX group level: the assignment is global. The ⟨*function*⟩ will not expand within an e-type or x-type argument.

4.3.3 Defining new functions using the signature

\cs_new:Nn
\cs_new:(cn|Ne|ce)

\cs_new:Nn ⟨*function*⟩ {⟨*code*⟩}

Creates ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed by the function. The definition is global and an error results if the ⟨*function*⟩ is already defined.

\cs_new_nopar:Nn
\cs_new_nopar:(cn|Ne|ce)

\cs_new_nopar:Nn ⟨*function*⟩ {⟨*code*⟩}

Creates ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain \par tokens. The definition is global and an error results if the ⟨*function*⟩ is already defined.

`\cs_new_protected:Nn`
`\cs_new_protected:(cn|Ne|ce)`

`\cs_new_protected:Nn` ⟨*function*⟩ `{`⟨*code*⟩`}`

Creates ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The ⟨*function*⟩ will not expand within an `e`-type or `x`-type argument. The definition is global and an error results if the ⟨*function*⟩ is already defined.

`\cs_new_protected_nopar:Nn`
`\cs_new_protected_nopar:(cn|Ne|ce)`

`\cs_new_protected_nopar:Nn` ⟨*function*⟩ `{`⟨*code*⟩`}`

Creates ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The ⟨*function*⟩ will not expand within an `e`-type or `x`-type argument. The definition is global and an error results if the ⟨*function*⟩ is already defined.

`\cs_set:Nn`
`\cs_set:(cn|Ne|ce)`

`\cs_set:Nn` ⟨*function*⟩ `{`⟨*code*⟩`}`

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the ⟨*function*⟩ is restricted to the current TeX group level.

`\cs_set_nopar:Nn`
`\cs_set_nopar:(cn|Ne|ce)`

`\cs_set_nopar:Nn` ⟨*function*⟩ `{`⟨*code*⟩`}`

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The assignment of a meaning to the ⟨*function*⟩ is restricted to the current TeX group level.

`\cs_set_protected:Nn`
`\cs_set_protected:(cn|Ne|ce)`

`\cs_set_protected:Nn` ⟨*function*⟩ `{`⟨*code*⟩`}`

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The ⟨*function*⟩ will not expand within an `e`-type or `x`-type argument. The assignment of a meaning to the ⟨*function*⟩ is restricted to the current TeX group level.

`\cs_set_protected_nopar:Nn`
`\cs_set_protected_nopar:(cn|Ne|ce)`

`\cs_set_protected_nopar:Nn` ⟨*function*⟩ `{`⟨*code*⟩`}`

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The ⟨*function*⟩ will not expand within an `e`-type or `x`-type argument. The assignment of a meaning to the ⟨*function*⟩ is restricted to the current TeX group level.

`\cs_gset:Nn`
`\cs_gset:(cn|Ne|ce)`

`\cs_gset:Nn` ⟨*function*⟩ {⟨*code*⟩}

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The assignment of a meaning to the ⟨*function*⟩ is global.

`\cs_gset_nopar:Nn`
`\cs_gset_nopar:(cn|Ne|ce)`

`\cs_gset_nopar:Nn` ⟨*function*⟩ {⟨*code*⟩}

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The assignment of a meaning to the ⟨*function*⟩ is global.

`\cs_gset_protected:Nn`
`\cs_gset_protected:(cn|Ne|ce)`

`\cs_gset_protected:Nn` ⟨*function*⟩ {⟨*code*⟩}

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. The ⟨*function*⟩ will not expand within an `e`-type or `x`-type argument. The assignment of a meaning to the ⟨*function*⟩ is global.

`\cs_gset_protected_nopar:Nn`
`\cs_gset_protected_nopar:(cn|Ne|ce)`

`\cs_gset_protected_nopar:Nn` ⟨*function*⟩ {⟨*code*⟩}

Sets ⟨*function*⟩ to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the number of ⟨*parameters*⟩ is detected automatically from the function signature. These ⟨*parameters*⟩ (`#1`, `#2`, *etc.*) will be replaced by those absorbed by the function. When the ⟨*function*⟩ is used the ⟨*parameters*⟩ absorbed cannot contain `\par` tokens. The ⟨*function*⟩ will not expand within an `e`-type or `x`-type argument. The assignment of a meaning to the ⟨*function*⟩ is global.

`\cs_generate_from_arg_count:NNnn`
`\cs_generate_from_arg_count:(NNno|cNnn|Ncnn)`

Updated: 2012-01-14

`\cs_generate_from_arg_count:NNnn` ⟨*function*⟩ ⟨*creator*⟩ {⟨*number*⟩} {⟨*code*⟩}

Uses the ⟨*creator*⟩ function (which should have signature `Npn`, for example `\cs_new:Npn`) to define a ⟨*function*⟩ which takes ⟨*number*⟩ arguments and has ⟨*code*⟩ as replacement text. The ⟨*number*⟩ of arguments is an integer expression, evaluated as detailed for `\int_eval:n`.

4.3.4 Copying control sequences

Control sequences (not just functions as defined above) can be set to have the same meaning using the functions described here. Making two control sequences equivalent means that the second control sequence is a *copy* of the first (rather than a pointer to it). Thus the old and new control sequence are not tied together: changes to one are not reflected in the other.

In the following text "cs" is used as an abbreviation for "control sequence".

\cs_new_eq:NN
\cs_new_eq:(Nc|cN|cc)

\cs_new_eq:NN ⟨*cs*$_1$⟩ ⟨*cs*$_2$⟩
\cs_new_eq:NN ⟨*cs*$_1$⟩ ⟨*token*⟩

Globally creates ⟨*control sequence*$_1$⟩ and sets it to have the same meaning as ⟨*control sequence*$_2$⟩ or ⟨*token*⟩. The second control sequence may subsequently be altered without affecting the copy.

\cs_set_eq:NN
\cs_set_eq:(Nc|cN|cc)

\cs_set_eq:NN ⟨*cs*$_1$⟩ ⟨*cs*$_2$⟩
\cs_set_eq:NN ⟨*cs*$_1$⟩ ⟨*token*⟩

Sets ⟨*control sequence*$_1$⟩ to have the same meaning as ⟨*control sequence*$_2$⟩ (or ⟨*token*⟩). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the ⟨*control sequence*$_1$⟩ is restricted to the current TEX group level.

\cs_gset_eq:NN
\cs_gset_eq:(Nc|cN|cc)

\cs_gset_eq:NN ⟨*cs*$_1$⟩ ⟨*cs*$_2$⟩
\cs_gset_eq:NN ⟨*cs*$_1$⟩ ⟨*token*⟩

Globally sets ⟨*control sequence*$_1$⟩ to have the same meaning as ⟨*control sequence*$_2$⟩ (or ⟨*token*⟩). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the ⟨*control sequence*$_1$⟩ is *not* restricted to the current TEX group level: the assignment is global.

4.3.5 Deleting control sequences

There are occasions where control sequences need to be deleted. This is handled in a very simple manner.

\cs_undefine:N
\cs_undefine:c
Updated: 2011-09-15

\cs_undefine:N ⟨*control sequence*⟩

Sets ⟨*control sequence*⟩ to be globally undefined.

4.3.6 Showing control sequences

\cs_meaning:N ⋆
\cs_meaning:c ⋆
Updated: 2011-12-22

\cs_meaning:N ⟨*control sequence*⟩

This function expands to the *meaning* of the ⟨*control sequence*⟩ control sequence. For a macro, this includes the ⟨*replacement text*⟩.

TEXhackers note: This is the TEX primitive \meaning. For tokens that are not control sequences, it is more logical to use \token_to_meaning:N. The c variant correctly reports undefined arguments.

\cs_show:N
\cs_show:c
Updated: 2017-02-14

\cs_show:N ⟨*control sequence*⟩

Displays the definition of the ⟨*control sequence*⟩ on the terminal.

TEXhackers note: This is similar to the TEX primitive \show, wrapped to a fixed number of characters per line.

\cs_log:N
\cs_log:c

New: 2014-08-22
Updated: 2017-02-14

\cs_log:N ⟨*control sequence*⟩

Writes the definition of the ⟨*control sequence*⟩ in the log file. See also \cs_show:N which displays the result in the terminal.

4.3.7 Converting to and from control sequences

\use:c ⋆

\use:c {⟨*control sequence name*⟩}

Expands the ⟨*control sequence name*⟩ until only characters remain, and then converts this into a control sequence. This process requires two expansions. As in other c-type arguments the ⟨*control sequence name*⟩ must, when fully expanded, consist of character tokens, typically a mixture of category code 10 (space), 11 (letter) and 12 (other).

As an example of the \use:c function, both

```
\use:c { a b c }
```

and

```
\tl_new:N  \l_my_tl
\tl_set:Nn \l_my_tl { a b c }
\use:c { \tl_use:N \l_my_tl }
```

would be equivalent to

```
\abc
```

after two expansions of \use:c.

\cs_if_exist_use:N ⋆
\cs_if_exist_use:c ⋆
\cs_if_exist_use:NTF ⋆
\cs_if_exist_use:cTF ⋆

New: 2012-11-10

\cs_if_exist_use:N ⟨*control sequence*⟩
\cs_if_exist_use:NTF ⟨*control sequence*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*control sequence*⟩ is currently defined according to the conditional \cs_if_exist:NTF (whether as a function or another control sequence type), and if it is inserts the ⟨*control sequence*⟩ into the input stream followed by the ⟨*true code*⟩. Otherwise the ⟨*false code*⟩ is used.

\cs:w ⋆
\cs_end: ⋆

\cs:w ⟨*control sequence name*⟩ \cs_end:

Converts the given ⟨*control sequence name*⟩ into a single control sequence token. This process requires one expansion. The content for ⟨*control sequence name*⟩ may be literal material or from other expandable functions. The ⟨*control sequence name*⟩ must, when fully expanded, consist of character tokens which are not active: typically of category code 10 (space), 11 (letter) or 12 (other), or a mixture of these.

TEXhackers note: These are the TEX primitives \csname and \endcsname.

As an example of the \cs:w and \cs_end: functions, both

```
\cs:w a b c \cs_end:
```

and

```
\tl_new:N  \l_my_tl
\tl_set:Nn \l_my_tl { a b c }
\cs:w \tl_use:N \l_my_tl \cs_end:
```

would be equivalent to

```
\abc
```

after one expansion of `\cs:w`.

`\cs_to_str:N` ★ `\cs_to_str:N` ⟨*control sequence*⟩

Converts the given ⟨*control sequence*⟩ into a series of characters with category code 12 (other), except spaces, of category code 10. The result does *not* include the current escape token, contrarily to `\token_to_str:N`. Full expansion of this function requires exactly 2 expansion steps, and so an `e`-type or `x`-type expansion, or two `o`-type expansions are required to convert the ⟨*control sequence*⟩ to a sequence of characters in the input stream. In most cases, an `f`-expansion is correct as well, but this loses a space at the start of the result.

4.4 Analysing control sequences

`\cs_split_function:N` ★ `\cs_split_function:N` ⟨*function*⟩

New: 2018-04-06

Splits the ⟨*function*⟩ into the ⟨*name*⟩ (*i.e.* the part before the colon) and the ⟨*signature*⟩ (*i.e.* after the colon). This information is then placed in the input stream in three parts: the ⟨*name*⟩, the ⟨*signature*⟩ and a logic token indicating if a colon was found (to differentiate variables from function names). The ⟨*name*⟩ does not include the escape character, and both the ⟨*name*⟩ and ⟨*signature*⟩ are made up of tokens with category code 12 (other).

The next three functions decompose TEX macros into their constituent parts: if the ⟨*token*⟩ passed is not a macro then no decomposition can occur. In the latter case, all three functions leave `\scan_stop:` in the input stream.

`\cs_prefix_spec:N` ★ `\cs_prefix_spec:N` ⟨*token*⟩

New: 2019-02-27

If the ⟨*token*⟩ is a macro, this function leaves the applicable TEX prefixes in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1~y #2 }
\cs_prefix_spec:N \next:nn
```

leaves `\long` in the input stream. If the ⟨*token*⟩ is not a macro then `\scan_stop:` is left in the input stream.

TEXhackers note: The prefix can be empty, `\long`, `\protected` or `\protected\long` with backslash replaced by the current escape character.

\cs_parameter_spec:N ⋆

New: 2022-06-24

\cs_parameter_spec:N ⟨*token*⟩

If the ⟨*token*⟩ is a macro, this function leaves the primitive TeX parameter specification in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1 y #2 }
\cs_parameter_spec:N \next:nn
```

leaves `#1#2` in the input stream. If the ⟨*token*⟩ is not a macro then `\scan_stop:` is left in the input stream.

TeXhackers note: If the parameter specification contains the string `->`, then the function produces incorrect results.

\cs_replacement_spec:N ⋆
\cs_replacement_spec:c ⋆

New: 2019-02-27

\cs_replacement_spec:N ⟨*token*⟩

If the ⟨*token*⟩ is a macro, this function leaves the replacement text in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1~y #2 }
\cs_replacement_spec:N \next:nn
```

leaves `x#1␣y#2` in the input stream. If the ⟨*token*⟩ is not a macro then `\scan_stop:` is left in the input stream.

TeXhackers note: If the parameter specification contains the string `->`, then the function produces incorrect results.

4.5 Using or removing tokens and arguments

Tokens in the input can be read and used or read and discarded. If one or more tokens are wrapped in braces then when absorbing them the outer set is removed. At the same time, the category code of each token is set when the token is read by a function (if it is read more than once, the category code is determined by the situation in force when first function absorbs the token).

```
\use:n    ★ \use:n    {⟨group1⟩}
\use:nn   ★ \use:nn   {⟨group1⟩} {⟨group2⟩}
\use:nnn  ★ \use:nnn  {⟨group1⟩} {⟨group2⟩} {⟨group3⟩}
\use:nnnn ★ \use:nnnn {⟨group1⟩} {⟨group2⟩} {⟨group3⟩} {⟨group4⟩}
```

As illustrated, these functions absorb between one and four arguments, as indicated by the argument specifier. The braces surrounding each argument are removed and the remaining tokens are left in the input stream. The category code of these tokens is also fixed by this process (if it has not already been by some other absorption). All of these functions require only a single expansion to operate, so that one expansion of

```
\use:nn { abc } { { def } }
```

results in the input stream containing

```
abc { def }
```

i.e. only the outer braces are removed.

TEXhackers note: The `\use:n` function is equivalent to LaTeX 2ε's `\@firstofone`.

\use_i:nn ⋆
\use_ii:nn ⋆
\use_i:nnn ⋆
\use_ii:nnn ⋆
\use_iii:nnn ⋆
\use_i:nnnn ⋆
\use_ii:nnnn ⋆
\use_iii:nnnn ⋆
\use_iv:nnnn ⋆
\use_i:nnnnn ⋆
\use_ii:nnnnn ⋆
\use_iii:nnnnn ⋆
\use_iv:nnnnn ⋆
\use_v:nnnnn ⋆
\use_i:nnnnnn ⋆
\use_ii:nnnnnn ⋆
\use_iii:nnnnnn ⋆
\use_iv:nnnnnn ⋆
\use_v:nnnnnn ⋆
\use_vi:nnnnnn ⋆
\use_i:nnnnnnn ⋆
\use_ii:nnnnnnn ⋆
\use_iii:nnnnnnn ⋆
\use_iv:nnnnnnn ⋆
\use_v:nnnnnnn ⋆
\use_vi:nnnnnnn ⋆
\use_vii:nnnnnnn ⋆
\use_i:nnnnnnnn ⋆
\use_ii:nnnnnnnn ⋆
\use_iii:nnnnnnnn ⋆
\use_iv:nnnnnnnn ⋆
\use_v:nnnnnnnn ⋆
\use_vi:nnnnnnnn ⋆
\use_vii:nnnnnnnn ⋆
\use_viii:nnnnnnnn ⋆
\use_i:nnnnnnnnn ⋆
\use_ii:nnnnnnnnn ⋆
\use_iii:nnnnnnnnn ⋆
\use_iv:nnnnnnnnn ⋆
\use_v:nnnnnnnnn ⋆
\use_vi:nnnnnnnnn ⋆
\use_vii:nnnnnnnnn ⋆
\use_viii:nnnnnnnnn ⋆
\use_ix:nnnnnnnnn ⋆

\use_i:nn {⟨*arg*$_1$⟩} {⟨*arg*$_2$⟩}
\use_i:nnn {⟨*arg*$_1$⟩} {⟨*arg*$_2$⟩} {⟨*arg*$_3$⟩}
\use_i:nnnn {⟨*arg*$_1$⟩} {⟨*arg*$_2$⟩} {⟨*arg*$_3$⟩} {⟨*arg*$_4$⟩}
\use_i:nnnnn {⟨*arg*$_1$⟩} {⟨*arg*$_2$⟩} {⟨*arg*$_3$⟩} {⟨*arg*$_4$⟩} {⟨*arg*$_5$⟩}
\use_i:nnnnnn {⟨*arg*$_1$⟩} {⟨*arg*$_2$⟩} {⟨*arg*$_3$⟩} {⟨*arg*$_4$⟩} {⟨*arg*$_5$⟩} {⟨*arg*$_6$⟩}
\use_i:nnnnnnn {⟨*arg*$_1$⟩} {⟨*arg*$_2$⟩} {⟨*arg*$_3$⟩} {⟨*arg*$_4$⟩} {⟨*arg*$_5$⟩} {⟨*arg*$_6$⟩} {⟨*arg*$_7$⟩}
\use_i:nnnnnnnn {⟨*arg*$_1$⟩} {⟨*arg*$_2$⟩} {⟨*arg*$_3$⟩} {⟨*arg*$_4$⟩} {⟨*arg*$_5$⟩} {⟨*arg*$_6$⟩} {⟨*arg*$_7$⟩} {⟨*arg*$_8$⟩}
\use_i:nnnnnnnnn {⟨*arg*$_1$⟩} {⟨*arg*$_2$⟩} {⟨*arg*$_3$⟩} {⟨*arg*$_4$⟩} {⟨*arg*$_5$⟩} {⟨*arg*$_6$⟩} {⟨*arg*$_7$⟩} {⟨*arg*$_8$⟩} {⟨*arg*$_9$⟩}

These functions absorb a number (n) arguments from the input stream. They then discard all arguments other than that indicated by the roman numeral, which is left in the input stream. For example, `\use_i:nn` discards the second argument, and leaves the content of the first argument in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

`\use_i_ii:nnn` ⋆

`\use_i_ii:nnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩}`

This function absorbs three arguments and leaves the content of the first and second in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect. An example:

```
\use_i_ii:nnn { abc } { { def } } { ghi }
```

results in the input stream containing

```
abc { def }
```

i.e. the outer braces are removed and the third group is removed.

`\use_ii_i:nn` ⋆

New: 2019-06-02

`\use_ii_i:nn {⟨arg₁⟩} {⟨arg₂⟩}`

This function absorbs two arguments and leaves the content of the second and first in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect.

`\use_none:n` ⋆
`\use_none:nn` ⋆
`\use_none:nnn` ⋆
`\use_none:nnnn` ⋆
`\use_none:nnnnn` ⋆
`\use_none:nnnnnn` ⋆
`\use_none:nnnnnnn` ⋆
`\use_none:nnnnnnnn` ⋆
`\use_none:nnnnnnnnn` ⋆

`\use_none:n {⟨group₁⟩}`

These functions absorb between one and nine groups from the input stream, leaving nothing on the resulting input stream. These functions work after a single expansion. One or more of the `n` arguments may be an unbraced single token (*i.e.* an `N` argument).

TEXhackers note: These are equivalent to LaTeX 2ε's `\@gobble`, `\@gobbletwo`, *etc.*

`\use:e` ⋆

New: 2018-06-18
Updated: 2023-07-05

`\use:e {⟨expandable tokens⟩}`

Fully expands the ⟨`token list`⟩ in an `e`-type manner, in which parameter character (usually `#`) need not be doubled, *and* the function remains fully expandable.

TEXhackers note: `\use:e` is a wrapper around the primitive `\expanded`. It requires two expansions to complete its action.

4.5.1 Selecting tokens from delimited arguments

A different kind of function for selecting tokens from the token stream are those that use delimited arguments.

`\use_none_delimit_by_q_nil:w` ⋆
`\use_none_delimit_by_q_stop:w` ⋆
`\use_none_delimit_by_q_recursion_stop:w` ⋆

`\use_none_delimit_by_q_nil:w ⟨balanced text⟩ \q_nil`
`\use_none_delimit_by_q_stop:w ⟨balanced text⟩ \q_stop`
`\use_none_delimit_by_q_recursion_stop:w ⟨balanced text⟩ \q_recursion_stop`

Absorb the ⟨`balanced text`⟩ from the input stream delimited by the marker given in the function name, leaving nothing in the input stream.

```
\use_i_delimit_by_q_nil:nw            ⋆
\use_i_delimit_by_q_stop:nw           ⋆
\use_i_delimit_by_q_recursion_stop:nw ⋆
```

\use_i_delimit_by_q_nil:nw {⟨*inserted tokens*⟩} ⟨*balanced text*⟩ \q_nil
\use_i_delimit_by_q_stop:nw {⟨*inserted tokens*⟩} ⟨*balanced text*⟩ \q_stop
\use_i_delimit_by_q_recursion_stop:nw {⟨*inserted tokens*⟩} ⟨*balanced text*⟩ \q_recursion_stop

Absorb the ⟨*balanced text*⟩ from the input stream delimited by the marker given in the function name, leaving ⟨*inserted tokens*⟩ in the input stream for further processing.

4.6 Predicates and conditionals

LaTeX3 has three concepts for conditional flow processing:

Branching conditionals Functions that carry out a test and then execute, depending on its result, either the code supplied as the ⟨*true code*⟩ or the ⟨*false code*⟩. These arguments are denoted with T and F, respectively. An example would be

\cs_if_free:cTF {abc} {⟨*true code*⟩} {⟨*false code*⟩}

a function that turns the first argument into a control sequence (since it's marked as c) then checks whether this control sequence is still free and then depending on the result carries out the code in the second argument (true case) or in the third argument (false case).

These type of functions are known as "conditionals"; whenever a TF function is defined it is usually accompanied by T and F functions as well. These are provided for convenience when the branch only needs to go a single way. Package writers are free to choose which types to define but the kernel definitions always provide all three versions.

Important to note is that these branching conditionals with ⟨*true code*⟩ and/or ⟨*false code*⟩ are always defined in a way that the code of the chosen alternative can operate on following tokens in the input stream.

These conditional functions may or may not be fully expandable, but if they are expandable they are accompanied by a "predicate" for the same test as described below.

Predicates "Predicates" are functions that return a special type of boolean value which can be tested by the boolean expression parser. All functions of this type are expandable and have names that end with _p in the description part. For example,

\cs_if_free_p:N

would be a predicate function for the same type of test as the conditional described above. It would return "true" if its argument (a single token denoted by N) is still free for definition. It would be used in constructions like

```
\bool_if:nTF
  { \cs_if_free_p:N \l_tmpz_tl || \cs_if_free_p:N \g_tmpz_tl }
  {⟨true code⟩} {⟨false code⟩}
```

For each predicate defined, a "branching conditional" also exists that behaves like a conditional described above.

Primitive conditionals There is a third variety of conditional, which is the original concept used in plain TeX and LaTeX 2ε. Their use is discouraged in expl3 (although still used in low-level definitions) because they are more fragile and in many cases require more expansion control (hence more code) than the two types of conditionals described above.

4.6.1 Tests on control sequences

\cs_if_eq_p:NN ⋆
\cs_if_eq_p:(Nc|cN|cc) ⋆
\cs_if_eq:NN*TF* ⋆
\cs_if_eq:(Nc|cN|cc)*TF* ⋆

\cs_if_eq_p:NN ⟨*cs*$_1$⟩ ⟨*cs*$_2$⟩
\cs_if_eq:NNTF ⟨*cs*$_1$⟩ ⟨*cs*$_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Compares the definition of two ⟨*control sequences*⟩ and is logically `true` if they are the same, *i.e.* if they have exactly the same definition when examined with `\cs_show:N`.

\cs_if_exist_p:N ⋆
\cs_if_exist_p:c ⋆
\cs_if_exist:N*TF* ⋆
\cs_if_exist:c*TF* ⋆

\cs_if_exist_p:N ⟨*control sequence*⟩
\cs_if_exist:NTF ⟨*control sequence*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*control sequence*⟩ is currently defined (whether as a function or another control sequence type). Any definition of ⟨*control sequence*⟩ other than `\relax` evaluates as `true`.

\cs_if_free_p:N ⋆
\cs_if_free_p:c ⋆
\cs_if_free:N*TF* ⋆
\cs_if_free:c*TF* ⋆

\cs_if_free_p:N ⟨*control sequence*⟩
\cs_if_free:NTF ⟨*control sequence*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*control sequence*⟩ is currently free to be defined. This test is `false` if the ⟨*control sequence*⟩ currently exists (as defined by `\cs_if_exist:NTF`).

4.6.2 Primitive conditionals

The ε-TeX engine itself provides many different conditionals. Some expand whatever comes after them and others don't. Hence the names for these underlying functions often contains a `:w` part but higher level functions are often available. See for instance `\int_compare_p:nNn` which is a wrapper for `\if_int_compare:w`.

Certain conditionals deal with specific data types like boxes and fonts and are described there. The ones described below are either the universal conditionals or deal with control sequences. We prefix primitive conditionals with `\if_`, except for `\if:w`.

\if_true: ⋆
\if_false: ⋆
\else: ⋆
\fi: ⋆
\reverse_if:N ⋆

\if_true: ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:
\if_false: ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:
\reverse_if:N ⟨*primitive conditional*⟩

`\if_true:` always executes ⟨*true code*⟩, while `\if_false:` always executes ⟨*false code*⟩. `\reverse_if:N` reverses any two-way primitive conditional. `\else:` and `\fi:` delimit the branches of the conditional. The function `\or:` is documented in l3int and used in case switches.

TeXhackers note: `\if_true:` and `\if_false:` are equivalent to their corresponding TeX primitive conditionals `\iftrue` and `\iffalse`; `\else:` and `\fi:` are the TeX primitives `\else` and `\fi`; `\reverse_if:N` is the ε-TeX primitive `\unless`.

`\if_meaning:w` ⋆ `\if_meaning:w` ⟨*arg*$_1$⟩ ⟨*arg*$_2$⟩ ⟨*true code*⟩ `\else:` ⟨*false code*⟩ `\fi:`

`\if_meaning:w` executes ⟨*true code*⟩ when ⟨*arg*$_1$⟩ and ⟨*arg*$_2$⟩ are the same, otherwise it executes ⟨*false code*⟩. ⟨*arg*$_1$⟩ and ⟨*arg*$_2$⟩ could be functions, variables, tokens; in all cases the *unexpanded* definitions are compared.

TEXhackers note: This is the TEX primitive `\ifx`.

`\if:w` ⋆
`\if_charcode:w` ⋆
`\if_catcode:w` ⋆

`\if:w` ⟨*token(s)*⟩ ⟨*true code*⟩ `\else:` ⟨*false code*⟩ `\fi:`
`\if_catcode:w` ⟨*token(s)*⟩ ⟨*true code*⟩ `\else:` ⟨*false code*⟩ `\fi:`

`\if_charcode:w` is an alternative name for `\if:w`. These conditionals expand ⟨*token(s)*⟩ until two unexpandable tokens ⟨*token*$_1$⟩ and ⟨*token*$_2$⟩ are found; any further tokens up to the next unbalanced `\else:` are the true branch, ending with ⟨*true code*⟩. It is executed if the condition is fulfilled, otherwise ⟨*false code*⟩ is executed. You can omit `\else:` when just in front of `\fi:` and you can nest `\if...\else:...\fi:` constructs inside the true branch or the ⟨*false code*⟩. With `\exp_not:N`, you can prevent the expansion of a token.

`\if_catcode:w` tests if ⟨*token*$_1$⟩ and ⟨*token*$_2$⟩ have the same category code whereas `\if:w` and `\if_charcode:w` test if they have the same character code.

TEXhackers note: `\if:w` and `\if_charcode:w` are both the TEX primitive `\if`. `\if_catcode:w` is the TEX primitive `\ifcat`.

`\if_cs_exist:N` ⋆
`\if_cs_exist:w` ⋆

`\if_cs_exist:N` ⟨*cs*⟩ ⟨*true code*⟩ `\else:` ⟨*false code*⟩ `\fi:`
`\if_cs_exist:w` ⟨*tokens*⟩ `\cs_end:` ⟨*true code*⟩ `\else:` ⟨*false code*⟩ `\fi:`

Check if ⟨*cs*⟩ appears in the hash table or if the control sequence that can be formed from ⟨*tokens*⟩ appears in the hash table. The latter function does not turn the control sequence in question into `\scan_stop:`! This can be useful when dealing with control sequences which cannot be entered as a single token.

TEXhackers note: These are the TEX primitives `\ifdefined` and `\ifcsname`.

`\if_mode_horizontal:` ⋆
`\if_mode_vertical:` ⋆
`\if_mode_math:` ⋆
`\if_mode_inner:` ⋆

`\if_mode_horizontal:` ⟨*true code*⟩ `\else:` ⟨*false code*⟩ `\fi:`

Execute ⟨*true code*⟩ if currently in horizontal mode, otherwise execute ⟨*false code*⟩. Similar for the other functions.

TEXhackers note: These are the TEX primitives `\ifhmode`, `\ifvmode`, `\ifmmode`, and `\ifinner`.

4.7 Starting a paragraph

\mode_leave_vertical:

New: 2017-07-04

`\mode_leave_vertical:`

Ensures that TeX is not in vertical (inter-paragraph) mode. In horizontal or math mode this command has no effect, in vertical mode it switches to horizontal mode, and inserts a box of width `\parindent`, followed by the `\everypar` token list.

TeXhackers note: This results in the contents of the `\everypar` token register being inserted, after `\mode_leave_vertical:` is complete. Notice that in contrast to the LaTeX 2_ε `\leavevmode` approach, no box is used by the method implemented here.

4.8 Debugging support

\debug_on:n
\debug_off:n

New: 2017-07-16
Updated: 2023-05-23

`\debug_on:n { ⟨comma-separated list⟩ }`
`\debug_off:n { ⟨comma-separated list⟩ }`

Turn on and off within a group various debugging code, some of which is also available as expl3 load-time options. The items that can be used in the ⟨*list*⟩ are

- `check-declarations` that checks all expl3 variables used were previously declared and that local/global variables (based on their name or on their first assignment) are only locally/globally assigned;
- `check-expressions` that checks integer, dimension, skip, and muskip expressions are not terminated prematurely;
- `deprecation` that makes deprecated commands produce errors;
- `log-functions` that logs function definitions and variable declarations;
- `all` that does all of the above.

Providing these as switches rather than options allows testing code even if it relies on other packages: load all other packages, call `\debug_on:n`, and load the code that one is interested in testing.

\debug_suspend:
\debug_resume:

New: 2017-11-28

`\debug_suspend: ... \debug_resume:`

Suppress (locally) errors and logging from `debug` commands, except for the `deprecation` errors. These pairs of commands can be nested. This can be used around pieces of code that are known to fail checks, if such failures should be ignored. See for instance l3cctab and l3coffins.

Chapter 5

The l3expan module Argument expansion

This module provides generic methods for expanding TEX arguments in a systematic manner. The functions in this module all have prefix `exp`.

Not all possible variations are implemented for every base function. Instead only those that are used within the LATEX3 kernel or otherwise seem to be of general interest are implemented. Consult the module description to find out which functions are actually defined. The next section explains how to define missing variants.

5.1 Defining new variants

The definition of variant forms for base functions may be necessary when writing new functions or when applying a kernel function in a situation that we haven't thought of before.

Internally preprocessing of arguments is done with functions of the form `\exp_-...`. They all look alike, an example would be `\exp_args:NNo`. This function has three arguments, the first and the second are a single tokens, while the third argument should be given in braces. Applying `\exp_args:NNo` expands the content of third argument once before any expansion of the first and second arguments. If `\seq_gpush:No` was not defined it could be coded in the following way:

```
\exp_args:NNo \seq_gpush:Nn
  \g_file_name_stack
  { \l_tmpa_tl }
```

In other words, the first argument to `\exp_args:NNo` is the base function and the other arguments are preprocessed and then passed to this base function. In the example the first argument to the base function should be a single token which is left unchanged while the second argument is expanded once. From this example we can also see how the variants are defined. They just expand into the appropriate `\exp_` function followed by the desired base function, *e.g.*

```
\cs_generate_variant:Nn \seq_gpush:Nn { No }
```

results in the definition of `\seq_gpush:No`

```
\cs_new:Npn \seq_gpush:No { \exp_args:NNo \seq_gpush:Nn }
```

Providing variants in this way in style files is safe as the `\cs_generate_variant:Nn` function will only create new definitions if there is not already one available. Therefore adding such definition to later releases of the kernel will not make such style files obsolete.

The steps above may be automated by using the function `\cs_generate_variant:Nn`, described next.

5.2 Methods for defining variants

We recall the set of available argument specifiers.

- `N` is used for single-token arguments while `c` constructs a control sequence from its name and passes it to a parent function as an `N`-type argument.
- Many argument types extract or expand some tokens and provide it as an `n`-type argument, namely a braced multiple-token argument: `V` extracts the value of a variable, `v` extracts the value from the name of a variable, `n` uses the argument as it is, `o` expands once, `f` expands fully the front of the token list, `e` and `x` expand fully all tokens (differences are explained later).
- A few odd argument types remain: `T` and `F` for conditional processing, otherwise identical to `n`-type arguments, `p` for the parameter text in definitions, `w` for arguments with a specific syntax, and `D` to denote primitives that should not be used directly.

\cs_generate_variant:Nn
\cs_generate_variant:cn

Updated: 2017-11-28

\cs_generate_variant:Nn ⟨*parent control sequence*⟩ {⟨*variant argument specifiers*⟩}

This function is used to define argument-specifier variants of the ⟨*parent control sequence*⟩ for LaTeX3 code-level macros. The ⟨*parent control sequence*⟩ is first separated into the ⟨*base name*⟩ and ⟨*original argument specifier*⟩. The comma-separated list of ⟨*variant argument specifiers*⟩ is then used to define variants of the ⟨*original argument specifier*⟩ if these are not already defined; entries which correspond to existing functions are silently ignored. For each ⟨*variant*⟩ given, a function is created that expands its arguments as detailed and passes them to the ⟨*parent control sequence*⟩. So for example

```
\cs_set:Npn \foo:Nn #1#2 { code here }
\cs_generate_variant:Nn \foo:Nn { c }
```

creates a new function `\foo:cn` which expands its first argument into a control sequence name and passes the result to `\foo:Nn`. Similarly

```
\cs_generate_variant:Nn \foo:Nn { NV , cV }
```

generates the functions `\foo:NV` and `\foo:cV` in the same way. The `\cs_generate_variant:Nn` function should only be applied if the ⟨*parent control sequence*⟩ is already defined. (This is only enforced if debugging support `check-declarations` is enabled.) If the ⟨*parent control sequence*⟩ is protected or if the ⟨*variant*⟩ involves any `x` argument, then the ⟨*variant control sequence*⟩ is also protected. The ⟨*variant*⟩ is created globally, as is any `\exp_args:N`⟨*variant*⟩ function needed to carry out the expansion. There is no need to re-apply `\cs_generate_variant:Nn` after changing the definition of the parent function: the variant will always use the current definition of the parent. Providing variants repeatedly is safe as `\cs_generate_variant:Nn` will only create new definitions if there is not already one available.

Only `n` and `N` arguments can be changed to other types. The only allowed changes are

- `c` variant of an `N` parent;
- `o`, `V`, `v`, `f`, `e`, or `x` variant of an `n` parent;
- `N`, `n`, `T`, `F`, or `p` argument unchanged.

This means the ⟨*parent*⟩ of a ⟨*variant*⟩ form is always unambiguous, even in cases where both an `n`-type parent and an `N`-type parent exist, such as for `\tl_count:n` and `\tl_count:N`.

When creating variants for conditional functions, `\prg_generate_conditional_variant:Nnn` provides a convenient way of handling the related function set.

For backward compatibility it is currently possible to make `n`, `o`, `V`, `v`, `f`, `e`, or `x`-type variants of an `N`-type argument or `N` or `c`-type variants of an `n`-type argument. Both are deprecated. The first because passing more than one token to an `N`-type argument will typically break the parent function's code. The second because programmers who use that most often want to access the value of a variable given its name, hence should use a `V`-type or `v`-type variant instead of `c`-type. In those cases, using the lower-level `\exp_args:No` or `\exp_args:Nc` functions explicitly is preferred to defining confusing variants.

\exp_args_generate:n \exp_args_generate:n {⟨*variant argument specifiers*⟩}

New: 2018-04-04 Updated: 2019-02-08

Defines \exp_args:N⟨*variant*⟩ functions for each ⟨*variant*⟩ given in the comma list {⟨*variant argument specifiers*⟩}. Each ⟨*variant*⟩ should consist of the letters N, c, n, V, v, o, f, e, x, p and the resulting function is protected if the letter x appears in the ⟨*variant*⟩. This is only useful for cases where \cs_generate_variant:Nn is not applicable.

5.3 Introducing the variants

The V type returns the value of a register, which can be one of tl, clist, int, skip, dim, muskip, or built-in TEX registers. The v type is the same except it first creates a control sequence out of its argument before returning the value.

In general, the programmer should not need to be concerned with expansion control. When simply using the content of a variable, functions with a V specifier should be used. For those referred to by (cs)name, the v specifier is available for the same purpose. Only when specific expansion steps are needed, such as when using delimited arguments, should the lower-level functions with o specifiers be employed.

The e type expands all tokens fully, starting from the first. More precisely the expansion is identical to that of TEX's \message (in particular # needs not be doubled). It relies on the primitive \expanded hence is fast.

The x type expands all tokens fully, starting from the first. In contrast to e, all macro parameter characters # must be doubled, and omitting this leads to low-level errors. In addition this type of expansion is not expandable, namely functions that have x in their signature do not themselves expand when appearing inside e or x expansion.

The f type is so special that it deserves an example. It is typically used in contexts where only expandable commands are allowed. Then x-expansion cannot be used, and f-expansion provides an alternative that expands the front of the token list as much as can be done in such contexts. For instance, say that we want to evaluate the integer expression $3 + 4$ and pass the result 7 as an argument to an expandable function \example:n. For this, one should define a variant using \cs_generate_variant:Nn \example:n { f }, then do

```
\example:f { \int_eval:n { 3 + 4 } }
```

Note that x-expansion would also expand \int_eval:n fully to its result 7, but the variant \example:x cannot be expandable. Note also that o-expansion would not expand \int_eval:n fully to its result since that function requires several expansions. Besides the fact that x-expansion is protected rather than expandable, another difference between f-expansion and x-expansion is that f-expansion expands tokens from the beginning and stops as soon as a non-expandable token is encountered, while x-expansion continues expanding further tokens. Thus, for instance

```
\example:f { \int_eval:n { 1 + 2 } , \int_eval:n { 3 + 4 } }
```

results in the call

```
\example:n { 3 , \int_eval:n { 3 + 4 } }
```

while using \example:x or \example:e instead results in

```
\example:n { 3 , 7 }
```

at the cost of being protected for x-type. If you use f type expansion in conditional processing then you should stick to using TF type functions only as the expansion does not finish any `\if... \fi:` itself!

It is important to note that both f- and o-type expansion are concerned with the expansion of tokens from left to right in their arguments. In particular, o-type expansion applies to the first *token* in the argument it receives: it is conceptually similar to

```
\exp_after:wN <base function> \exp_after:wN { <argument> }
```

At the same time, f-type expansion stops at the *first* non-expandable token. This means for example that both

```
\tl_set:No \l_tmpa_tl { { \g_tmpb_tl } }
```

and

```
\tl_set:Nf \l_tmpa_tl { { \g_tmpb_tl } }
```

leave `\g_tmpb_tl` unchanged: { is the first token in the argument and is non-expandable.

It is usually best to keep the following in mind when using variant forms.

- Variants with x-type arguments (that are fully expanded before being passed to the n-type base function) are never expandable even when the base function is. Such variants cannot work correctly in arguments that are themselves subject to expansion. Consider using f or e expansion.
- In contrast, e expansion (full expansion, almost like x except for the treatment of #) does not prevent variants from being expandable (if the base function is).
- Finally f expansion only expands the front of the token list, stopping at the first non-expandable token. This may fail to fully expand the argument.

When speed is essential (for functions that do very little work and whose variants are used numerous times in a document) the following considerations apply because the speed of internal functions that expand the arguments of a base function depend on what needs doing with each argument and where this happens in the list of arguments:

- for fastest processing any c-type arguments should come first followed by all other modified arguments;
- unchanged N-type args that appear before modified ones have a small performance hit;
- unchanged n-type args that appear before modified ones have a relative larger performance hit.

5.4 Manipulating the first argument

These functions are described in detail: expansion of multiple tokens follows the same rules but is described in a shorter fashion.

`\exp_args:Nc` ⋆
`\exp_args:cc` ⋆

`\exp_args:Nc` ⟨*function*⟩ {⟨*tokens*⟩}

This function absorbs two arguments (the ⟨*function*⟩ name and the ⟨*tokens*⟩). The ⟨*tokens*⟩ are expanded until only characters remain, and are then turned into a control sequence. The result is inserted into the input stream *after* reinsertion of the ⟨*function*⟩. Thus the ⟨*function*⟩ may take more than one argument: all others are left unchanged.

The `:cc` variant constructs the ⟨*function*⟩ name in the same manner as described for the ⟨*tokens*⟩.

`\exp_args:No` ⋆

`\exp_args:No` ⟨*function*⟩ {⟨*tokens*⟩} ...

This function absorbs two arguments (the ⟨*function*⟩ name and the ⟨*tokens*⟩). The ⟨*tokens*⟩ are expanded once, and the result is inserted in braces into the input stream *after* reinsertion of the ⟨*function*⟩. Thus the ⟨*function*⟩ may take more than one argument: all others are left unchanged.

`\exp_args:NV` ⋆

`\exp_args:NV` ⟨*function*⟩ ⟨*variable*⟩

This function absorbs two arguments (the names of the ⟨*function*⟩ and the ⟨*variable*⟩). The content of the ⟨*variable*⟩ are recovered and placed inside braces into the input stream *after* reinsertion of the ⟨*function*⟩. Thus the ⟨*function*⟩ may take more than one argument: all others are left unchanged.

`\exp_args:Nv` ⋆

`\exp_args:Nv` ⟨*function*⟩ {⟨*tokens*⟩}

This function absorbs two arguments (the ⟨*function*⟩ name and the ⟨*tokens*⟩). The ⟨*tokens*⟩ are expanded until only characters remain, and are then turned into a control sequence. This control sequence should be the name of a ⟨*variable*⟩. The content of the ⟨*variable*⟩ are recovered and placed inside braces into the input stream *after* reinsertion of the ⟨*function*⟩. Thus the ⟨*function*⟩ may take more than one argument: all others are left unchanged.

`\exp_args:Ne` ⋆

New: 2018-05-15

`\exp_args:Ne` ⟨*function*⟩ {⟨*tokens*⟩}

This function absorbs two arguments (the ⟨*function*⟩ name and the ⟨*tokens*⟩) and exhaustively expands the ⟨*tokens*⟩. The result is inserted in braces into the input stream *after* reinsertion of the ⟨*function*⟩. Thus the ⟨*function*⟩ may take more than one argument: all others are left unchanged.

`\exp_args:Nf` ⋆

`\exp_args:Nf` ⟨*function*⟩ {⟨*tokens*⟩}

This function absorbs two arguments (the ⟨*function*⟩ name and the ⟨*tokens*⟩). The ⟨*tokens*⟩ are fully expanded until the first non-expandable token is found (if that is a space it is removed), and the result is inserted in braces into the input stream *after* reinsertion of the ⟨*function*⟩. Thus the ⟨*function*⟩ may take more than one argument: all others are left unchanged.

`\exp_args:Nx`

`\exp_args:Nx` ⟨*function*⟩ {⟨*tokens*⟩}

This function absorbs two arguments (the ⟨*function*⟩ name and the ⟨*tokens*⟩) and exhaustively expands the ⟨*tokens*⟩. The result is inserted in braces into the input stream *after* reinsertion of the ⟨*function*⟩. Thus the ⟨*function*⟩ may take more than one argument: all others are left unchanged.

5.5 Manipulating two arguments

\exp_args:NNc ⋆
\exp_args:NNo ⋆
\exp_args:NNV ⋆
\exp_args:NNv ⋆
\exp_args:NNe ⋆
\exp_args:NNf ⋆
\exp_args:Ncc ⋆
\exp_args:Nco ⋆
\exp_args:NcV ⋆
\exp_args:Ncv ⋆
\exp_args:Ncf ⋆
\exp_args:NVV ⋆

Updated: 2018-05-15

`\exp_args:NNc` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ `{`⟨*tokens*⟩`}`

These optimized functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments.

\exp_args:Nnc ⋆
\exp_args:Nno ⋆
\exp_args:NnV ⋆
\exp_args:Nnv ⋆
\exp_args:Nne ⋆
\exp_args:Nnf ⋆
\exp_args:Noc ⋆
\exp_args:Noo ⋆
\exp_args:Nof ⋆
\exp_args:NVo ⋆
\exp_args:Nfo ⋆
\exp_args:Nff ⋆
\exp_args:Nee ⋆

Updated: 2018-05-15

`\exp_args:Noo` ⟨*token*⟩ `{`⟨*tokens*$_1$⟩`}` `{`⟨*tokens*$_2$⟩`}`

These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments.

\exp_args:NNx
\exp_args:Ncx
\exp_args:Nnx
\exp_args:Nox
\exp_args:Nxo
\exp_args:Nxx

`\exp_args:NNx` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ `{`⟨*tokens*⟩`}`

These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments. These functions are not expandable due to their `x`-type argument.

5.6 Manipulating three arguments

\exp_args:NNNo ⋆
\exp_args:NNNV ⋆
\exp_args:NNNv ⋆
\exp_args:NNNe ⋆
\exp_args:Nccc ⋆
\exp_args:NcNc ⋆
\exp_args:NcNo ⋆
\exp_args:Ncco ⋆

`\exp_args:NNNo` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ ⟨*token*$_3$⟩ `{`⟨*tokens*⟩`}`

These optimized functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, *etc.*

\exp_args:NNcf ⋆
\exp_args:NNno ⋆
\exp_args:NNnV ⋆
\exp_args:NNoo ⋆
\exp_args:NNVV ⋆
\exp_args:Ncno ⋆
\exp_args:NcnV ⋆
\exp_args:Ncoo ⋆
\exp_args:NcVV ⋆
\exp_args:Nnnc ⋆
\exp_args:Nnno ⋆
\exp_args:Nnnf ⋆
\exp_args:Nnff ⋆
\exp_args:Nooo ⋆
\exp_args:Noof ⋆
\exp_args:Nffo ⋆
\exp_args:Neee ⋆

\exp_args:NNoo ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ {⟨*token*$_3$⟩} {⟨*tokens*⟩}

These functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, *etc.*

\exp_args:NNNx
\exp_args:NNnx
\exp_args:NNox
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Noox

New: 2015-08-12

\exp_args:NNnx ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ {⟨*tokens*$_1$⟩} {⟨*tokens*$_2$⟩}

These functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, *etc.*

5.7 Unbraced expansion

\exp_last_unbraced:No ⋆
\exp_last_unbraced:NV ⋆
\exp_last_unbraced:Nv ⋆
\exp_last_unbraced:Ne ⋆
\exp_last_unbraced:Nf ⋆
\exp_last_unbraced:NNo ⋆
\exp_last_unbraced:NNV ⋆
\exp_last_unbraced:NNf ⋆
\exp_last_unbraced:Nco ⋆
\exp_last_unbraced:NcV ⋆
\exp_last_unbraced:Nno ⋆
\exp_last_unbraced:Nnf ⋆
\exp_last_unbraced:Noo ⋆
\exp_last_unbraced:Nfo ⋆
\exp_last_unbraced:NNNo ⋆
\exp_last_unbraced:NNNV ⋆
\exp_last_unbraced:NNNf ⋆
\exp_last_unbraced:NnNo ⋆
\exp_last_unbraced:NNNNo ⋆
\exp_last_unbraced:NNNNf ⋆

Updated: 2018-05-15

\exp_last_unbraced:Nno ⟨*token*⟩ {⟨*tokens*$_1$⟩} {⟨*tokens*$_2$⟩}

These functions absorb the number of arguments given by their specification, carry out the expansion indicated and leave the results in the input stream, with the last argument not surrounded by the usual braces. Of these, the :Nno, :Noo, :Nfo and :NnNo variants need slower processing.

TEXhackers note: As an optimization, the last argument is unbraced by some of those functions before expansion. This can cause problems if the argument is empty: for instance, \exp_last_unbraced:Nf \foo_bar:w { } \q_stop leads to an infinite loop, as the quark is f-expanded.

\exp_last_unbraced:Nx

\exp_last_unbraced:Nx ⟨*function*⟩ {⟨*tokens*⟩}

This function fully expands the ⟨*tokens*⟩ and leaves the result in the input stream after reinsertion of the ⟨*function*⟩. This function is not expandable.

\exp_last_two_unbraced:Noo ⋆

\exp_last_two_unbraced:Noo ⟨*token*⟩ {⟨*tokens*$_1$⟩} {⟨*tokens*$_2$⟩}

This function absorbs three arguments and expands the second and third once. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments, which are not wrapped in braces. This function needs special (slower) processing.

\exp_after:wN ⋆

\exp_after:wN ⟨*token*$_1$⟩ ⟨*token*$_2$⟩

Carries out a single expansion of ⟨*token*$_2$⟩ (which may consume arguments) prior to the expansion of ⟨*token*$_1$⟩. If ⟨*token*$_2$⟩ has no expansion (for example, if it is a character) then it is left unchanged. It is important to notice that ⟨*token*$_1$⟩ may be *any* single token, including group-opening and -closing tokens ({ or } assuming normal TEX category codes). Unless specifically required this should be avoided: expansion should be carried out using an appropriate argument specifier variant or the appropriate \exp_args:N⟨*variant*⟩ function.

TEXhackers note: This is the TEX primitive \expandafter.

5.8 Preventing expansion

Despite the fact that the following functions are all about preventing expansion, they're designed to be used in an expandable context and hence are all marked as being 'expandable' since they themselves disappear after the expansion has completed.

\exp_not:N ⋆

\exp_not:N ⟨token⟩

Prevents expansion of the ⟨token⟩ in a context where it would otherwise be expanded, for example an e-type or x-type argument or the first token in an o-type or f-type argument.

TEXhackers note: This is the TEX primitive \noexpand. It only prevents expansion. At the beginning of an f-type argument, a space ⟨token⟩ is removed even if it appears as \exp_not:N \c_space_token. In an e-expanding definition (\cs_new:Npe), a macro parameter introduces an argument even if it appears as \exp_not:N # 1. This differs from \exp_not:n.

\exp_not:c ⋆

\exp_not:c {⟨tokens⟩}

Expands the ⟨tokens⟩ until only characters remain, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited using \exp_not:N.

\exp_not:n ⋆

\exp_not:n {⟨tokens⟩}

Prevents expansion of the ⟨tokens⟩ in an e-type or x-type argument. In all other cases the ⟨tokens⟩ continue to be expanded, for example in the input stream or in other types of arguments such as c, f, v. The argument of \exp_not:n *must* be surrounded by braces.

TEXhackers note: This is the ε-TEX primitive \unexpanded. In an e-expanding definition (\cs_new:Npe), \exp_not:n {#1} is equivalent to ##1 rather than to #1, namely it inserts the two characters # and 1, and \exp_not:n {#} is equivalent to #, namely it inserts the character #.

\exp_not:o ⋆

\exp_not:o {⟨tokens⟩}

Expands the ⟨tokens⟩ once, then prevents any further expansion in e-type or x-type arguments using \exp_not:n.

\exp_not:V ⋆

\exp_not:V ⟨variable⟩

Recovers the content of the ⟨variable⟩, then prevents expansion of this material in e-type or x-type arguments using \exp_not:n.

\exp_not:v ⋆

\exp_not:v {⟨tokens⟩}

Expands the ⟨tokens⟩ until only characters remains, and then converts this into a control sequence which should be a ⟨variable⟩ name. The content of the ⟨variable⟩ is recovered, and further expansion in e-type or x-type arguments is prevented using \exp_not:n.

\exp_not:e ⋆ \exp_not:e {⟨tokens⟩}

Expands ⟨*tokens*⟩ exhaustively, then protects the result of the expansion (including any tokens which were not expanded) from further expansion in e-type or x-type arguments using `\exp_not:n`. This is very rarely useful but is provided for consistency.

\exp_not:f ⋆ \exp_not:f {⟨tokens⟩}

Expands ⟨*tokens*⟩ fully until the first unexpandable token is found (if it is a space it is removed). Expansion then stops, and the result of the expansion (including any tokens which were not expanded) is protected from further expansion in e-type or x-type arguments using `\exp_not:n`.

\exp_stop_f: ⋆ \foo_bar:f { ⟨tokens⟩ \exp_stop_f: ⟨more tokens⟩ }

Updated: 2011-06-03

This function terminates an f-type expansion. Thus if a function `\foo_bar:f` starts an f-type expansion and all of ⟨*tokens*⟩ are expandable `\exp_stop_f:` terminates the expansion of tokens even if ⟨*more tokens*⟩ are also expandable. The function itself is an implicit space token. Inside an e-type or x-type expansion, it retains its form, but when typeset it produces the underlying space (␣).

5.9 Controlled expansion

The expl3 language makes all efforts to hide the complexity of TeX expansion from the programmer by providing concepts that evaluate/expand arguments of functions prior to calling the "base" functions. Thus, instead of using many `\expandafter` calls and other trickery it is usually a matter of choosing the right variant of a function to achieve a desired result.

Of course, deep down TeX is using expansion as always and there are cases where a programmer needs to control that expansion directly; typical situations are basic data manipulation tools. This section documents the functions for that level. These commands are used throughout the kernel code, but we hope that outside the kernel there will be little need to resort to them. Instead the argument manipulation methods document above should usually be sufficient.

While `\exp_after:wN` expands one token (out of order) it is sometimes necessary to expand several tokens in one go. The next set of commands provide this functionality. Be aware that it is absolutely required that the programmer has full control over the tokens to be expanded, i.e., it is not possible to use these functions to expand unknown input as part of ⟨*expandable-tokens*⟩ as that will break badly if unexpandable tokens are encountered in that place!

`\exp:w` ⋆
`\exp_end:` ⋆

New: 2015-08-23

`\exp:w` ⟨*expandable tokens*⟩ `\exp_end:`

Expands ⟨*expandable-tokens*⟩ until reaching `\exp_end:` at which point expansion stops. The full expansion of ⟨*expandable tokens*⟩ has to be empty. If any token in ⟨*expandable tokens*⟩ or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result `\exp_end:` will be misinterpreted later on.[3]

In typical use cases the `\exp_end:` is hidden somewhere in the replacement text of ⟨*expandable-tokens*⟩ rather than being on the same expansion level than `\exp:w`, e.g., you may see code such as

```
\exp:w \@@_case:NnTF #1 {#2} { } { }
```

where somewhere during the expansion of `\@@_case:NnTF` the `\exp_end:` gets generated.

TEXhackers note: The current implementation uses `\romannumeral` hence ignores space tokens and explicit signs `+` and `-` in the expansion of the ⟨*expandable tokens*⟩, but this should not be relied upon.

`\exp:w` ⋆
`\exp_end_continue_f:w` ⋆

New: 2015-08-23

`\exp:w` ⟨*expandable-tokens*⟩ `\exp_end_continue_f:w` ⟨*further-tokens*⟩

Expands ⟨*expandable-tokens*⟩ until reaching `\exp_end_continue_f:w` at which point expansion continues as an `f`-type expansion expanding ⟨*further-tokens*⟩ until an unexpandable token is encountered (or the `f`-type expansion is explicitly terminated by `\exp_stop_f:`). As with all `f`-type expansions a space ending the expansion gets removed.

The full expansion of ⟨*expandable-tokens*⟩ has to be empty. If any token in ⟨*expandable-tokens*⟩ or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result `\exp_end_continue_f:w` will be misinterpreted later on.[4]

In typical use cases ⟨*expandable-tokens*⟩ contains no tokens at all, e.g., you will see code such as

```
\exp_after:wN { \exp:w \exp_end_continue_f:w #2 }
```

where the `\exp_after:wN` triggers an `f`-expansion of the tokens in `#2`. For technical reasons this has to happen using two tokens (if they would be hidden inside another command `\exp_after:wN` would only expand the command but not trigger any additional `f`-expansion).

You might wonder why there are two different approaches available, after all the effect of

`\exp:w` ⟨*expandable-tokens*⟩ `\exp_end:`

can be alternatively achieved through an `f`-type expansion by using `\exp_stop_f:`, i.e.

`\exp:w \exp_end_continue_f:w` ⟨*expandable-tokens*⟩ `\exp_stop_f:`

The reason is simply that the first approach is slightly faster (one less token to parse and less expansion internally) so in places where such performance really matters and where we want to explicitly stop the expansion at a defined point the first form is preferable.

[3]Due to the implementation you might get the character in position 0 in the current font (typically "`") in the output without any error message!

\exp:w ⋆
\exp_end_continue_f:nw ⋆

New: 2015-08-23

`\exp:w` ⟨*expandable-tokens*⟩ `\exp_end_continue_f:nw` ⟨*further-tokens*⟩

The difference to `\exp_end_continue_f:w` is that we first we pick up an argument which is then returned to the input stream. If ⟨*further-tokens*⟩ starts with space tokens then these space tokens are removed while searching for the argument. If it starts with a brace group then the braces are removed. Thus such spaces or braces will not terminate the `f`-type expansion.

5.10 Internal functions

\::n
\::N
\::p
\::c
\::o
\::e
\::f
\::x
\::v
\::V
\:::

`\cs_new:Npn \exp_args:Ncof { \::c \::o \::f \::: }`

Internal forms for the base expansion types. These names do *not* conform to the general LaTeX3 approach as this makes them more readily visible in the log and so forth. They should not be used outside this module.

\::o_unbraced
\::e_unbraced
\::f_unbraced
\::x_unbraced
\::v_unbraced
\::V_unbraced

`\cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \::: }`

Internal forms for the expansion types which leave the terminal argument unbraced. These names do *not* conform to the general LaTeX3 approach as this makes them more readily visible in the log and so forth. They should not be used outside this module.

[4] In this particular case you may get a character into the output as well as an error message.

Chapter 6

The l3sort module
Sorting functions

6.1 Controlling sorting

LaTeX3 comes with a facility to sort list variables (sequences, token lists, or comma-lists) according to some user-defined comparison. For instance,

```
\clist_set:Nn \l_foo_clist { 3 , 01 , -2 , 5 , +1 }
\clist_sort:Nn \l_foo_clist
  {
    \int_compare:nNnTF { #1 } > { #2 }
      { \sort_return_swapped: }
      { \sort_return_same: }
  }
```

results in `\l_foo_clist` holding the values `{ -2 , 01 , +1 , 3 , 5 }` sorted in non-decreasing order.

The code defining the comparison should call `\sort_return_swapped:` if the two items given as `#1` and `#2` are not in the correct order, and otherwise it should call `\sort_return_same:` to indicate that the order of this pair of items should not be changed.

For instance, a ⟨*comparison code*⟩ consisting only of `\sort_return_same:` with no test yields a trivial sort: the final order is identical to the original order. Conversely, using a ⟨*comparison code*⟩ consisting only of `\sort_return_swapped:` reverses the list (in a fairly inefficient way).

TeXhackers note: The current implementation is limited to sorting approximately 20000 items (40000 in LuaTeX), depending on what other packages are loaded.

Internally, the code from l3sort stores items in `\toks` registers allocated locally. Thus, the ⟨*comparison code*⟩ should not call `\newtoks` or other commands that allocate new `\toks` registers. On the other hand, altering the value of a previously allocated `\toks` register is not a problem.

\sort_return_same:
\sort_return_swapped:

New: 2017-02-06

`\seq_sort:Nn ⟨seq var⟩`
`{ ... \sort_return_same: or \sort_return_swapped: ... }`

Indicates whether to keep the order or swap the order of two items that are compared in the sorting code. Only one of the `\sort_return_...` functions should be used by the code, according to the results of some tests on the items `#1` and `#2` to be compared.

Chapter 7

The l3tl-analysis module Analysing token lists

This module provides functions that are particularly useful in the l3regex module for mapping through a token list one ⟨*token*⟩ at a time (including begin-group/end-group tokens). For `\tl_analysis_map_inline:Nn` or `\tl_analysis_map_inline:nn`, the token list is given as an argument; the analogous function `\peek_analysis_map_inline:n` documented in l3token finds tokens in the input stream instead. In both cases the user provides ⟨*inline code*⟩ that receives three arguments for each ⟨*token*⟩:

- ⟨*tokens*⟩, which both o-expand and e/x-expand to the ⟨*token*⟩. The detailed form of ⟨*tokens*⟩ may change in later releases.
- ⟨*char code*⟩, a decimal representation of the character code of the ⟨*token*⟩, -1 if it is a control sequence.
- ⟨*catcode*⟩, a capital hexadecimal digit which denotes the category code of the ⟨*token*⟩ (0: control sequence, 1: begin-group, 2: end-group, 3: math shift, 4: alignment tab, 6: parameter, 7: superscript, 8: subscript, A: space, B: letter, C: other, D: active). This can be converted to an integer by writing "⟨*catcode*⟩.

In addition, there is a debugging function `\tl_analysis_show:n`, very similar to the `\ShowTokens` macro from the ted package.

`\tl_analysis_show:N`
`\tl_analysis_show:n`
`\tl_analysis_log:N`
`\tl_analysis_log:n`

New: 2021-05-11

`\tl_analysis_show:n {⟨token list⟩}`
`\tl_analysis_log:n {⟨token list⟩}`

Displays to the terminal (or log) the detailed decomposition of the ⟨*token list*⟩ into tokens, showing the category code of each character token, the meaning of control sequences and active characters, and the value of registers.

`\tl_analysis_map_inline:nn`
`\tl_analysis_map_inline:Nn`

New: 2018-04-09
Updated: 2022-03-26

`\tl_analysis_map_inline:nn {⟨token list⟩} {⟨inline function⟩}`

Applies the ⟨*inline function*⟩ to each individual ⟨*token*⟩ in the ⟨*token list*⟩. The ⟨*inline function*⟩ receives three arguments as explained above. As all other mappings the mapping is done at the current group level, *i.e.* any local assignments made by the ⟨*inline function*⟩ remain in effect after the loop.

Chapter 8

The l3regex module Regular expressions in TEX

The l3regex module provides regular expression testing, extraction of submatches, splitting, and replacement, all acting on token lists. The syntax of regular expressions is mostly a subset of the PCRE syntax (and very close to POSIX), with some additions due to the fact that TEX manipulates tokens rather than characters. For performance reasons, only a limited set of features are implemented. Notably, back-references are not supported.

Let us give a few examples. After

```
\tl_set:Nn \l_my_tl { That~cat. }
\regex_replace_once:nnN { at } { is } \l_my_tl
```

the token list variable `\l_my_tl` holds the text "`This cat.`", where the first occurrence of "`at`" was replaced by "`is`". A more complicated example is a pattern to emphasize each word and add a comma after it:

```
\regex_replace_all:nnN { \w+ } { \c{emph}\cB\{ \0 \cE\} , } \l_my_tl
```

The `\w` sequence represents any "word" character, and `+` indicates that the `\w` sequence should be repeated as many times as possible (at least once), hence matching a word in the input token list. In the replacement text, `\0` denotes the full match (here, a word). The command `\emph` is inserted using `\c{emph}`, and its argument `\0` is put between braces `\cB\{` and `\cE\}`.

If a regular expression is to be used several times, it can be compiled once, and stored in a regex variable using `\regex_set:Nn`. For example,

```
\regex_new:N \l_foo_regex
\regex_set:Nn \l_foo_regex { \c{begin} \cB. (\c[^BE].*) \cE. }
```

stores in `\l_foo_regex` a regular expression which matches the starting marker for an environment: `\begin`, followed by a begin-group token (`\cB.`), then any number of tokens which are neither begin-group nor end-group character tokens (`\c[^BE].*`), ending with an end-group token (`\cE.`). As explained in the next section, the parentheses "capture" the result of `\c[^BE].*`, giving us access to the name of the environment when doing replacements.

8.1 Syntax of regular expressions

8.1.1 Regular expression examples

We start with a few examples, and encourage the reader to apply `\regex_show:n` to these regular expressions.

- `Cat` matches the word "Cat" capitalized in this way, but also matches the beginning of the word "Cattle": use `\bCat\b` to match a complete word only.
- `[abc]` matches one letter among "a", "b", "c"; the pattern `(a|b|c)` matches the same three possible letters (but see the discussion of submatches below).
- `[A-Za-z]*` matches any number (due to the quantifier `*`) of Latin letters (not accented).
- `\c{[A-Za-z]*}` matches a control sequence made of Latin letters.
- `\_[^\_]*\_` matches an underscore, any number of characters other than underscore, and another underscore; it is equivalent to `\_.*?\_` where `.` matches arbitrary characters and the lazy quantifier `*?` means to match as few characters as possible, thus avoiding matching underscores.
- `[\+\-]?\d+` matches an explicit integer with at most one sign.
- `[\+\-\␣]*\d+\␣*` matches an explicit integer with any number of + and − signs, with spaces allowed except within the mantissa, and surrounded by spaces.
- `[\+\-\␣]*(\d+|\d*\.\d+)\␣*` matches an explicit integer or decimal number; using `[.,]` instead of `\.` would allow the comma as a decimal marker.
- `[\+\-\␣]*(\d+|\d*\.\d+)\␣*((?i)pt|in|[cem]m|ex|[bs]p|[dn]d|[pcn]c)\␣*` matches an explicit dimension with any unit that TeX knows, where `(?i)` means to treat lowercase and uppercase letters identically.
- `[\+\-\␣]*((?i)nan|inf|(\d+|\d*\.\d+)(\␣*e[\+\-\␣]*\d+)?)\␣*` matches an explicit floating point number or the special values `nan` and `inf` (with signs and spaces allowed).
- `[\+\-\␣]*(\d+|\cC.)\␣*` matches an explicit integer or control sequence (without checking whether it is an integer variable).
- `\G.*?\K` at the beginning of a regular expression matches and discards (due to `\K`) everything between the end of the previous match (`\G`) and what is matched by the rest of the regular expression; this is useful in `\regex_replace_all:nnN` when the goal is to extract matches or submatches in a finer way than with `\regex_extract_all:nnN`.

While it is impossible for a regular expression to match only integer expressions, `[\+\-\(]*\d+\)*([\+\-*/][\+\-\(]*\d+\)*)*` matches among other things all valid integer expressions (made only with explicit integers). One should follow it with further testing.

8.1.2 Characters in regular expressions

Most characters match exactly themselves, with an arbitrary category code. Some characters are special and must be escaped with a backslash (*e.g.*, `\*` matches a star character). Some escape sequences of the form backslash–letter also have a special meaning (for instance `\d` matches any digit). As a rule,

- every alphanumeric character (`A`–`Z`, `a`–`z`, `0`–`9`) matches exactly itself, and should not be escaped, because `\A`, `\B`, ... have special meanings;
- non-alphanumeric printable ascii characters can (and should) always be escaped: many of them have special meanings (*e.g.*, use `\(`, `\)`, `\?`, `\.`, `\^`);
- spaces should always be escaped (even in character classes);
- any other character may be escaped or not, without any effect: both versions match exactly that character.

Note that these rules play nicely with the fact that many non-alphanumeric characters are difficult to input into TeX under normal category codes. For instance, `\\abc\%` matches the characters `\abc%` (with arbitrary category codes), but does not match the control sequence `\abc` followed by a percent character. Matching control sequences can be done using the `\c{`⟨*regex*⟩`}` syntax (see below).

Any special character which appears at a place where its special behaviour cannot apply matches itself instead (for instance, a quantifier appearing at the beginning of a string), after raising a warning.

Characters.

`\x{hh...}` Character with hex code `hh...`

`\xhh` Character with hex code `hh`.

`\a` Alarm (hex 07).

`\e` Escape (hex 1B).

`\f` Form-feed (hex 0C).

`\n` New line (hex 0A).

`\r` Carriage return (hex 0D).

`\t` Horizontal tab (hex 09).

8.1.3 Characters classes

Character properties.

`.` A single period matches any token.

`\d` Any decimal digit.

`\h` Any horizontal space character, equivalent to `[\ \^^I]`: space and tab.

`\s` Any space character, equivalent to `[\ \^^I\^^J\^^L\^^M]`.

- `\v` Any vertical space character, equivalent to `[\^^J\^^K\^^L\^^M]`. Note that `\^^K` is a vertical space, but not a space, for compatibility with Perl.
- `\w` Any word character, *i.e.*, alphanumerics and underscore, equivalent to the explicit class `[A-Za-z0-9\_]`.
- `\D` Any token not matched by `\d`.
- `\H` Any token not matched by `\h`.
- `\N` Any token other than the `\n` character (hex 0A).
- `\S` Any token not matched by `\s`.
- `\V` Any token not matched by `\v`.
- `\W` Any token not matched by `\w`.

Of those, `.`, `\D`, `\H`, `\N`, `\S`, `\V`, and `\W` match arbitrary control sequences.

Character classes match exactly one token in the subject.

- `[...]` Positive character class. Matches any of the specified tokens.
- `[^...]` Negative character class. Matches any token other than the specified characters.
- `[x-y]` Within a character class, this denotes a range (can be used with escaped characters).
- `[:⟨name⟩:]` Within a character class (one more set of brackets), this denotes the POSIX character class ⟨*name*⟩, which can be `alnum`, `alpha`, `ascii`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `word`, or `xdigit`.
- `[:^⟨name⟩:]` Negative POSIX character class.

For instance, `[a-oq-z\cC.]` matches any lowercase latin letter except `p`, as well as control sequences (see below for a description of `\c`).

In character classes, only `[`, `^`, `-`, `]`, `\` and spaces are special, and should be escaped. Other non-alphanumeric characters can still be escaped without harm. Any escape sequence which matches a single character (`\d`, `\D`, *etc.*) is supported in character classes. If the first character is `^`, then the meaning of the character class is inverted; `^` appearing anywhere else in the range is not special. If the first character (possibly following a leading `^`) is `]` then it does not need to be escaped since ending the range there would make it empty. Ranges of characters can be expressed using `-`, for instance, `[\D 0-5]` and `[^6-9]` are equivalent.

8.1.4 Structure: alternatives, groups, repetitions

Quantifiers (repetition).

- `?` 0 or 1, greedy.
- `??` 0 or 1, lazy.
- `*` 0 or more, greedy.
- `*?` 0 or more, lazy.
- `+` 1 or more, greedy.

`+?` 1 or more, lazy.

`{`n`}` Exactly n.

`{`n`,}` n or more, greedy.

`{`n`,}?` n or more, lazy.

`{`n, m`}` At least n, no more than m, greedy.

`{`n, m`}?` At least n, no more than m, lazy.

For greedy quantifiers the regex code will first investigate matches that involve as many repetitions as possible, while for lazy quantifiers it investigates matches with as few repetitions as possible first.

Alternation and capturing groups.

`A|B|C` Either one of `A`, `B`, or `C`, investigating `A` first.

`(...)` Capturing group.

`(?:...)` Non-capturing group.

`(?|...)` Non-capturing group which resets the group number for capturing groups in each alternative. The following group is numbered with the first unused group number.

Capturing groups are a means of extracting information about the match. Parenthesized groups are labelled in the order of their opening parenthesis, starting at 1. The contents of those groups corresponding to the "best" match (leftmost longest) can be extracted and stored in a sequence of token lists using for instance `\regex_extract_once:nnNTF`.

The `\K` escape sequence resets the beginning of the match to the current position in the token list. This only affects what is reported as the full match. For instance,

```
\regex_extract_all:nnN { a \K . } { a123aaxyz } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{1}` and `{a}`: the true matches are `{a1}` and `{aa}`, but they are trimmed by the use of `\K`. The `\K` command does not affect capturing groups: for instance,

```
\regex_extract_once:nnN { (. \K c)+ \d } { acbc3 } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{c3}` and `{bc}`: the true match is `{acbc3}`, with first submatch `{bc}`, but `\K` resets the beginning of the match to the last position where it appears.

8.1.5 Matching exact tokens

The `\c` escape sequence allows to test the category code of tokens, and match control sequences. Each character category is represented by a single uppercase letter:

- `C` for control sequences;
- `B` for begin-group tokens;
- `E` for end-group tokens;

- `M` for math shift;
- `T` for alignment tab tokens;
- `P` for macro parameter tokens;
- `U` for superscript tokens (up);
- `D` for subscript tokens (down);
- `S` for spaces;
- `L` for letters;
- `O` for others; and
- `A` for active characters.

The `\c` escape sequence is used as follows.

`\c{⟨regex⟩}` A control sequence whose csname matches the ⟨*regex*⟩, anchored at the beginning and end, so that `\c{begin}` matches exactly `\begin`, and nothing else.

`\cX` Applies to the next object, which can be a character, escape character sequence such as `\x{0A}`, character class, or group, and forces this object to only match tokens with category `X` (any of `CBEMTPUDSLOA`. For instance, `\cL[A-Z\d]` matches uppercase letters and digits of category code letter, `\cC.` matches any control sequence, and `\cO(abc)` matches `abc` where each character has category other.[5]

`\c[XYZ]` Applies to the next object, and forces it to only match tokens with category `X`, `Y`, or `Z` (each being any of `CBEMTPUDSLOA`). For instance, `\c[LSO](..)` matches two tokens of category letter, space, or other.

`\c[^XYZ]` Applies to the next object and prevents it from matching any token with category `X`, `Y`, or `Z` (each being any of `CBEMTPUDSLOA`). For instance, `\c[^O]\d` matches digits which have any category different from other.

The category code tests can be used inside classes; for instance, `[\cO\d \c[LO][A-F]]` matches what TeX considers as hexadecimal digits, namely digits with category other, or uppercase letters from `A` to `F` with category either letter or other. Within a group affected by a category code test, the outer test can be overridden by a nested test: for instance, `\cL(ab\cO\*cd)` matches `ab*cd` where all characters are of category letter, except `*` which has category other.

The `\u` escape sequence allows to insert the contents of a token list directly into a regular expression or a replacement, avoiding the need to escape special characters. Namely, `\u{⟨var name⟩}` matches the exact contents (both character codes and category codes) of the variable `\⟨var name⟩`, which are obtained by applying `\exp_not:v {⟨var name⟩}` at the time the regular expression is compiled. Within a `\c{...}` control sequence matching, the `\u` escape sequence only expands its argument once, in effect performing `\tl_to_str:v`. Quantifiers are supported.

The `\ur` escape sequence allows to insert the contents of a `regex` variable into a larger regular expression. For instance, `A\ur{l_tmpa_regex}D` matches the tokens `A` and

[5]This last example also captures "`abc`" as a regex group; to avoid this use a non-capturing group `\cO(?:abc)`.

D separated by something that matches the regular expression `\l_tmpa_regex`. This behaves as if a non-capturing group were surrounding `\l_tmpa_regex`, and any group contained in `\l_tmpa_regex` is converted to a non-capturing group. Quantifiers are supported.

For instance, if `\l_tmpa_regex` has value `B|C`, then `A\ur{l_tmpa_regex}D` is equivalent to `A(?:B|C)D` (matching `ABD` or `ACD`) and not to `AB|CD` (matching `AB` or `CD`). To get the latter effect, it is simplest to use TeX's expansion machinery directly: if `\l_mymodule_BC_tl` contains `B|C` then the following two lines show the same result:

```
\regex_show:n { A \u{l_mymodule_BC_tl} D }
\regex_show:n { A B | C D }
```

8.1.6 Miscellaneous

Anchors and simple assertions.

`\b` Word boundary: either the previous token is matched by `\w` and the next by `\W`, or the opposite. For this purpose, the ends of the token list are considered as `\W`.

`\B` Not a word boundary: between two `\w` tokens or two `\W` tokens (including the boundary).

`^`or `\A` Start of the subject token list.

`$`, `\Z` or `\z` End of the subject token list.

`\G` Start of the current match. This is only different from `^` in the case of multiple matches: for instance `\regex_count:nnN { \G a } { aaba } \l_tmpa_int` yields 2, but replacing `\G` by `^` would result in `\l_tmpa_int` holding the value 1.

The option `(?i)` makes the match case insensitive (treating `A`–`Z` and `a`–`z` as equivalent, with no support yet for Unicode case changing). This applies until the end of the group in which it appears, and can be reverted using `(?-i)`. For instance, in `(?i)(a(?-i)b|c)d`, the letters `a` and `d` are affected by the `i` option. Characters within ranges and classes are affected individually: `(?i)[\?-B]` is equivalent to `[\?@ABab]` (and differs from the much larger class `[\?-b]`), and `(?i)[^aeiou]` matches any character which is not a vowel. The `i` option has no effect on `\c{...}`, on `\u{...}`, on character properties, or on character classes, for instance it has no effect at all in `(?i)\u{l_foo_tl}\d\d[[:lower:]]`.

8.2 Syntax of the replacement text

Most of the features described in regular expressions do not make sense within the replacement text. Backslash introduces various special constructions, described further below:

- `\0` is the whole match;
- `\1` is the submatch that was matched by the first (capturing) group `(...)`; similarly for `\2`, ..., `\9` and `\g{⟨number⟩}`;
- `\␣` inserts a space (spaces are ignored when not escaped);

- `\a`, `\e`, `\f`, `\n`, `\r`, `\t`, `\xhh`, `\x{hhh}` correspond to single characters as in regular expressions;
- `\c{`⟨*cs name*⟩`}` inserts a control sequence;
- `\c`⟨*category*⟩⟨*character*⟩ (see below);
- `\u{`⟨*tl var name*⟩`}` inserts the contents of the ⟨*tl var*⟩ (see below).

Characters other than backslash and space are simply inserted in the result (but since the replacement text is first converted to a string, one should also escape characters that are special for TeX, for instance use `\#`). Non-alphanumeric characters can always be safely escaped with a backslash.

For instance,

```
\tl_set:Nn \l_my_tl { Hello,~world! }
\regex_replace_all:nnN { ([er]?l|o) . } { (\0--\1) } \l_my_tl
```

results in `\l_my_tl` holding `H(ell--el)(o,--o) w(or--o)(ld--l)!`

The submatches are numbered according to the order in which the opening parenthesis of capturing groups appear in the regular expression to match. The n-th submatch is empty if there are fewer than n capturing groups or for capturing groups that appear in alternatives that were not used for the match. In case a capturing group matches several times during a match (due to quantifiers) only the last match is used in the replacement text. Submatches always keep the same category codes as in the original token list.

By default, the category code of characters inserted by the replacement are determined by the prevailing category code regime at the time where the replacement is made, with two exceptions:

- space characters (with character code 32) inserted with `\␣` or `\x20` or `\x{20}` have category code 10 regardless of the prevailing category code regime;
- if the category code would be 0 (escape), 5 (newline), 9 (ignore), 14 (comment) or 15 (invalid), it is replaced by 12 (other) instead.

The escape sequence `\c` allows to insert characters with arbitrary category codes, as well as control sequences.

`\cX(...)` Produces the characters "..." with category `X`, which must be one of `CBEMTPUDSLOA` as in regular expressions. Parentheses are optional for a single character (which can be an escape sequence). When nested, the innermost category code applies, for instance `\cL(Hello\cS\ world)!` gives this text with standard category codes.

`\c{`⟨*text*⟩`}` Produces the control sequence with csname ⟨*text*⟩. The ⟨*text*⟩ may contain references to the submatches `\0`, `\1`, and so on, as in the example for `\u` below.

The escape sequence `\u{`⟨*var name*⟩`}` allows to insert the contents of the variable with name ⟨*var name*⟩ directly into the replacement, giving an easier control of category codes. When nested in `\c{...}` and `\u{...}` constructions, the `\u` and `\c` escape sequences perform `\tl_to_str:v`, namely extract the value of the control sequence and turn it into a string. Matches can also be used within the arguments of `\c` and `\u`. For instance,

```
\tl_set:Nn \l_my_one_tl { first }
\tl_set:Nn \l_my_two_tl { \emph{second} }
\tl_set:Nn \l_my_tl { one , two , one , one }
\regex_replace_all:nnN { [^,]+ } { \u{l_my_\0_tl} } \l_my_tl
```

results in `\l_my_tl` holding `first,\emph{second},first,first`.

Regex replacement is also a convenient way to produce token lists with arbitrary category codes. For instance

```
\tl_clear:N \l_tmpa_tl
\regex_replace_all:nnN { } { \cU\% \cA\~ } \l_tmpa_tl
```

results in `\l_tmpa_tl` containing the percent character with category code 7 (superscript) and an active tilde character.

8.3 Pre-compiling regular expressions

If a regular expression is to be used several times, it is better to compile it once rather than doing it each time the regular expression is used. The compiled regular expression is stored in a variable. All of the l3regex module's functions can be given their regular expression argument either as an explicit string or as a compiled regular expression.

`\regex_new:N`

New: 2017-05-26

`\regex_new:N ⟨regex var⟩`

Creates a new ⟨*regex var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*regex var*⟩ is initially such that it never matches.

`\regex_set:Nn`
`\regex_gset:Nn`

New: 2017-05-26

`\regex_set:Nn ⟨regex var⟩ {⟨regex⟩}`

Stores a compiled version of the ⟨*regular expression*⟩ in the ⟨*regex var*⟩. The assignment is local for `\regex_set:Nn` and global for `\regex_gset:Nn`. For instance, this function can be used as

```
\regex_new:N \l_my_regex
\regex_set:Nn \l_my_regex { my\ (simple\ )? reg(ex|ular\ expression) }
```

`\regex_const:Nn`

New: 2017-05-26

`\regex_const:Nn ⟨regex var⟩ {⟨regex⟩}`

Creates a new constant ⟨*regex var*⟩ or raises an error if the name is already taken. The value of the ⟨*regex var*⟩ is set globally to the compiled version of the ⟨*regular expression*⟩.

`\regex_show:N`
`\regex_show:n`
`\regex_log:N`
`\regex_log:n`

New: 2021-04-26
Updated: 2021-04-29

`\regex_show:n {⟨regex⟩}`
`\regex_log:n {⟨regex⟩}`

Displays in the terminal or writes in the log file (respectively) how l3regex interprets the ⟨*regex*⟩. For instance, `\regex_show:n {\A X|Y}` shows

```
+-branch
  anchor at start (\A)
  char code 88 (X)
+-branch
  char code 89 (Y)
```

indicating that the anchor `\A` only applies to the first branch: the second branch is not anchored to the beginning of the match.

8.4 Matching

All regular expression functions are available in both `:n` and `:N` variants. The former require a "standard" regular expression, while the later require a compiled expression as generated by `\regex_set:Nn`.

`\regex_match:nnTF`
`\regex_match:nVTF`
`\regex_match:NnTF`
`\regex_match:NVTF`

New: 2017-05-26

`\regex_match:nnTF {⟨regex⟩} {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}`

Tests whether the ⟨*regular expression*⟩ matches any part of the ⟨*token list*⟩. For instance,

```
\regex_match:nnTF { b [cde]* } { abecdcx } { TRUE } { FALSE }
\regex_match:nnTF { [b-dq-w] } { example } { TRUE } { FALSE }
```

leaves `TRUE` then `FALSE` in the input stream.

`\regex_count:nnN`
`\regex_count:nVN`
`\regex_count:NnN`
`\regex_count:NVN`

New: 2017-05-26

`\regex_count:nnN {⟨regex⟩} {⟨token list⟩} ⟨int var⟩`

Sets ⟨*int var*⟩ within the current TeX group level equal to the number of times ⟨*regular expression*⟩ appears in ⟨*token list*⟩. The search starts by finding the left-most longest match, respecting greedy and lazy (non-greedy) operators. Then the search starts again from the character following the last character of the previous match, until reaching the end of the token list. Infinite loops are prevented in the case where the regular expression can match an empty token list: then we count one match between each pair of characters. For instance,

```
\int_new:N \l_foo_int
\regex_count:nnN { (b+|c) } { abbababcbb } \l_foo_int
```

results in `\l_foo_int` taking the value 5.

`\regex_match_case:nn`
`\regex_match_case:nnTF`

New: 2022-01-10

```
\regex_match_case:nnTF
  {
    {⟨regex_1⟩} {⟨code case_1⟩}
    {⟨regex_2⟩} {⟨code case_2⟩}
    ...
    {⟨regex_n⟩} {⟨code case_n⟩}
  } {⟨token list⟩}
  {⟨true code⟩} {⟨false code⟩}
```

Determines which of the ⟨*regular expressions*⟩ matches at the earliest point in the ⟨*token list*⟩, and leaves the corresponding ⟨*code*⟩ followed by the ⟨*true code*⟩ in the input stream. If several ⟨*regex*⟩ match starting at the same point, then the first one in the list is selected and the others are discarded. If none of the ⟨*regex*⟩ match, the ⟨*false code*⟩ is left in the input stream. Each ⟨*regex*⟩ can either be given as a regex variable or as an explicit regular expression.

In detail, for each starting position in the ⟨*token list*⟩, each of the ⟨*regex*⟩ is searched in turn. If one of them matches then the corresponding ⟨*code*⟩ is used and everything else is discarded, while if none of the ⟨*regex*⟩ match at a given position then the next starting position is attempted. If none of the ⟨*regex*⟩ match anywhere in the ⟨*token list*⟩ then nothing is left in the input stream. Note that this differs from nested `\regex_match:nnTF` statements since all ⟨*regex*⟩ are attempted at each position rather than attempting to match ⟨*regex*$_1$⟩ at every position before moving on to ⟨*regex*$_2$⟩.

8.5 Submatch extraction

\regex_extract_once:nnN
\regex_extract_once:nVN
\regex_extract_once:nnNTF
\regex_extract_once:nVNTF
\regex_extract_once:NnN
\regex_extract_once:NVN
\regex_extract_once:NnNTF
\regex_extract_once:NVNTF

New: 2017-05-26

\regex_extract_once:nnN {⟨*regex*⟩} {⟨*token list*⟩} ⟨*seq var*⟩
\regex_extract_once:nnNTF {⟨*regex*⟩} {⟨*token list*⟩} ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Finds the first match of the ⟨*regular expression*⟩ in the ⟨*token list*⟩. If it exists, the match is stored as the first item of the ⟨*seq var*⟩, and further items are the contents of capturing groups, in the order of their opening parenthesis. The ⟨*seq var*⟩ is assigned locally. If there is no match, the ⟨*seq var*⟩ is cleared. The testing versions insert the ⟨*true code*⟩ into the input stream if a match was found, and the ⟨*false code*⟩ otherwise.

For instance, assume that you type

```
\regex_extract_once:nnNTF { \A(La)?TeX(!*)\Z } { LaTeX!!! } \l_foo_seq
  { true } { false }
```

Then the regular expression (anchored at the start with `\A` and at the end with `\Z`) must match the whole token list. The first capturing group, `(La)?`, matches `La`, and the second capturing group, `(!*)`, matches `!!!`. Thus, `\l_foo_seq` contains as a result the items `{LaTeX!!!}`, `{La}`, and `{!!!}`, and the `true` branch is left in the input stream. Note that the n-th item of `\l_foo_seq`, as obtained using `\seq_item:Nn`, correspond to the submatch numbered $(n-1)$ in functions such as `\regex_replace_once:nnN`.

\regex_extract_all:nnN
\regex_extract_all:nVN
\regex_extract_all:nnNTF
\regex_extract_all:nVNTF
\regex_extract_all:NnN
\regex_extract_all:NVN
\regex_extract_all:NnNTF
\regex_extract_all:NVNTF

New: 2017-05-26

\regex_extract_all:nnN {⟨*regex*⟩} {⟨*token list*⟩} ⟨*seq var*⟩
\regex_extract_all:nnNTF {⟨*regex*⟩} {⟨*token list*⟩} ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Finds all matches of the ⟨*regular expression*⟩ in the ⟨*token list*⟩, and stores all the submatch information in a single sequence (concatenating the results of multiple `\regex_extract_once:nnN` calls). The ⟨*seq var*⟩ is assigned locally. If there is no match, the ⟨*seq var*⟩ is cleared. The testing versions insert the ⟨*true code*⟩ into the input stream if a match was found, and the ⟨*false code*⟩ otherwise. For instance, assume that you type

```
\regex_extract_all:nnNTF { \w+ } { Hello,~world! } \l_foo_seq
  { true } { false }
```

Then the regular expression matches twice, the resulting sequence contains the two items `{Hello}` and `{world}`, and the `true` branch is left in the input stream.

\regex_split:nnN
\regex_split:nVN
\regex_split:nnNTF
\regex_split:nVNTF
\regex_split:NnN
\regex_split:NVN
\regex_split:NnNTF
\regex_split:NVNTF

New: 2017-05-26

\regex_split:nnN {⟨*regular expression*⟩} {⟨*token list*⟩} ⟨*seq var*⟩
\regex_split:nnNTF {⟨*regular expression*⟩} {⟨*token list*⟩} ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Splits the ⟨*token list*⟩ into a sequence of parts, delimited by matches of the ⟨*regular expression*⟩. If the ⟨*regular expression*⟩ has capturing groups, then the token lists that they match are stored as items of the sequence as well. The assignment to ⟨*seq var*⟩ is local. If no match is found the resulting ⟨*seq var*⟩ has the ⟨*token list*⟩ as its sole item. If the ⟨*regular expression*⟩ matches the empty token list, then the ⟨*token list*⟩ is split into single tokens. The testing versions insert the ⟨*true code*⟩ into the input stream if a match was found, and the ⟨*false code*⟩ otherwise. For example, after

```
\seq_new:N \l_path_seq
\regex_split:nnNTF { / } { the/path/for/this/file.tex } \l_path_seq
  { true } { false }
```

the sequence `\l_path_seq` contains the items `{the}`, `{path}`, `{for}`, `{this}`, and `{file.tex}`, and the `true` branch is left in the input stream.

8.6 Replacement

\regex_replace_once:nnN
\regex_replace_once:nVN
\regex_replace_once:nnNTF
\regex_replace_once:nVNTF
\regex_replace_once:NnN
\regex_replace_once:NVN
\regex_replace_once:NnNTF
\regex_replace_once:NVNTF

New: 2017-05-26

\regex_replace_once:nnN {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*tl var*⟩
\regex_replace_once:nnNTF {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Searches for the ⟨*regular expression*⟩ in the contents of the ⟨*tl var*⟩ and replaces the first match with the ⟨*replacement*⟩. In the ⟨*replacement*⟩, `\0` represents the full match, `\1` represent the contents of the first capturing group, `\2` of the second, *etc.* The result is assigned locally to ⟨*tl var*⟩.

\regex_replace_all:nnN
\regex_replace_all:nVN
\regex_replace_all:nnNTF
\regex_replace_all:nVNTF
\regex_replace_all:NnN
\regex_replace_all:NVN
\regex_replace_all:NnNTF
\regex_replace_all:NVNTF

New: 2017-05-26

\regex_replace_all:nnN {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*tl var*⟩
\regex_replace_all:nnNTF {⟨*regular expression*⟩} {⟨*replacement*⟩} ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Replaces all occurrences of the ⟨*regular expression*⟩ in the contents of the ⟨*tl var*⟩ by the ⟨*replacement*⟩, where `\0` represents the full match, `\1` represent the contents of the first capturing group, `\2` of the second, *etc.* Every match is treated independently, and matches cannot overlap. The result is assigned locally to ⟨*tl var*⟩.

`\regex_replace_case_once:nN`
`\regex_replace_case_once:nNTF`

New: 2022-01-10

```
\regex_replace_case_once:nNTF
  {
    {⟨regex1⟩} {⟨replacement1⟩}
    {⟨regex2⟩} {⟨replacement2⟩}
    ...
    {⟨regexn⟩} {⟨replacementn⟩}
  } ⟨tl var⟩
  {⟨true code⟩} {⟨false code⟩}
```

Replaces the earliest match of the regular expression `(?|`⟨*regex*$_1$⟩`|`...`|`⟨*regex*$_n$⟩`)` in the ⟨*token list variable*⟩ by the ⟨*replacement*⟩ corresponding to which ⟨*regex*$_i$⟩ matched, then leaves the ⟨*true code*⟩ in the input stream. If none of the ⟨*regex*⟩ match, then the ⟨*tl var*⟩ is not modified, and the ⟨*false code*⟩ is left in the input stream. Each ⟨*regex*⟩ can either be given as a regex variable or as an explicit regular expression.

In detail, for each starting position in the ⟨*token list*⟩, each of the ⟨*regex*⟩ is searched in turn. If one of them matches then it is replaced by the corresponding ⟨*replacement*⟩ as described for `\regex_replace_once:nnN`. This is equivalent to checking with `\regex_match_case:nn` which ⟨*regex*⟩ matches, then performing the replacement with `\regex_replace_once:nnN`.

`\regex_replace_case_all:nN`
`\regex_replace_case_all:nNTF`

New: 2022-01-10

```
\regex_replace_case_all:nNTF
  {
    {⟨regex1⟩} {⟨replacement1⟩}
    {⟨regex2⟩} {⟨replacement2⟩}
    ...
    {⟨regexn⟩} {⟨replacementn⟩}
  } ⟨tl var⟩
  {⟨true code⟩} {⟨false code⟩}
```

Replaces all occurrences of all ⟨*regex*⟩ in the ⟨*token list*⟩ by the corresponding ⟨*replacement*⟩. Every match is treated independently, and matches cannot overlap. The result is assigned locally to ⟨*tl var*⟩, and the ⟨*true code*⟩ or ⟨*false code*⟩ is left in the input stream depending on whether any replacement was made or not.

In detail, for each starting position in the ⟨*token list*⟩, each of the ⟨*regex*⟩ is searched in turn. If one of them matches then it is replaced by the corresponding ⟨*replacement*⟩, and the search resumes at the position that follows this match (and replacement). For instance

```
\tl_set:Nn \l_tmpa_tl { Hello,~world! }
\regex_replace_case_all:nN
  {
    { [A-Za-z]+ } { ``\0'' }
    { \b } { --- }
    { . } { [\0] }
  } \l_tmpa_tl
```

results in `\l_tmpa_tl` having the contents ``` ``Hello''---[,][␣]``world''---[!] ```. Note in particular that the word-boundary assertion `\b` did not match at the start of words because the case `[A-Za-z]+` matched at these positions. To change this, one could simply swap the order of the two cases in the argument of `\regex_replace_case_all:nN`.

8.7 Scratch regular expressions

`\l_tmpa_regex`
`\l_tmpb_regex`
New: 2017-12-11

Scratch regex for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_regex`
`\g_tmpb_regex`
New: 2017-12-11

Scratch regex for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

8.8 Bugs, misfeatures, future work, and other possibilities

The following need to be done now.

- Rewrite the documentation in a more ordered way, perhaps add a BNF?

  Additional error-checking to come.

- Clean up the use of messages.
- Cleaner error reporting in the replacement phase.
- Add tracing information.
- Detect attempts to use back-references and other non-implemented syntax.
- Test for the maximum register `\c_max_register_int`.
- Find out whether the fact that `\W` and friends match the end-marker leads to bugs. Possibly update `\__regex_item_reverse:n`.
- The empty cs should be matched by `\c{}`, not by `\c{csname.?endcsname\s?}`.

  Code improvements to come.

- Shift arrays so that the useful information starts at position 1.
- Only build `\c{...}` once.
- Use arrays for the left and right state stacks when compiling a regex.
- Should `\__regex_action_free_group:n` only be used for greedy `{n,}` quantifier? (I think not.)
- Quantifiers for `\u` and assertions.
- When matching, keep track of an explicit stack of `curr_state` and `curr_submatches`.
- If possible, when a state is reused by the same thread, kill other subthreads.

- Use an array rather than `\g__regex_balance_tl` to build the function `\__regex_-replacement_balance_one_match:n`.
- Reduce the number of epsilon-transitions in alternatives.
- Optimize simple strings: use less states (`abcade` should give two states, for `abc` and `ade`). [Does that really make sense?]
- Optimize groups with no alternative.
- Optimize states with a single `\__regex_action_free:n`.
- Optimize the use of `\__regex_action_success:` by inserting it in state 2 directly instead of having an extra transition.
- Optimize the use of `\int_step_...` functions.
- Groups don't capture within regexes for csnames; optimize and document.
- Better "show" for anchors, properties, and catcode tests.
- Does `\K` really need a new state for itself?
- When compiling, use a boolean `in_cs` and less magic numbers.

The following features are likely to be implemented at some point in the future.

- General look-ahead/behind assertions.
- Regex matching on external files.
- Conditional subpatterns with look ahead/behind: "if what follows is [...], then [...]".
- `(*..)` and `(?..)` sequences to set some options.
- UTF-8 mode for pdfTeX.
- Newline conventions are not done. In particular, we should have an option for `.` not to match newlines. Also, `\A` should differ from `^`, and `\Z`, `\z` and `$` should differ.
- Unicode properties: `\p{..}` and `\P{..}`; `\X` which should match any "extended" Unicode sequence. This requires to manipulate a lot of data, probably using tree-boxes.

The following features of PCRE or Perl may or may not be implemented.

- Callout with `(?C...)` or other syntax: some internal code changes make that possible, and it can be useful for instance in the replacement code to stop a regex replacement when some marker has been found; this raises the question of a potential `\regex_break:` and then of playing well with `\tl_map_break:` called from within the code in a regex. It also raises the question of nested calls to the regex machinery, which is a problem since `\fontdimen` are global.
- Conditional subpatterns (other than with a look-ahead or look-behind condition): this is non-regular, isn't it?

- Named subpatterns: TEX programmers have lived so far without any need for named macro parameters.

The following features of PCRE or Perl will definitely not be implemented.

- Back-references: non-regular feature, this requires backtracking, which is prohibitively slow.
- Recursion: this is a non-regular feature.
- Atomic grouping, possessive quantifiers: those tools, mostly meant to fix catastrophic backtracking, are unnecessary in a non-backtracking algorithm, and difficult to implement.
- Subroutine calls: this syntactic sugar is difficult to include in a non-backtracking algorithm, in particular because the corresponding group should be treated as atomic.
- Backtracking control verbs: intrinsically tied to backtracking.
- `\ddd`, matching the character with octal code `ddd`: we already have `\x{...}` and the syntax is confusingly close to what we could have used for backreferences (`\1`, `\2`, ...), making it harder to produce useful error message.
- `\cx`, similar to TEX's own `\^^x`.
- Comments: TEX already has its own system for comments.
- `\Q...\E` escaping: this would require to read the argument verbatim, which is not in the scope of this module.
- `\C` single byte in UTF-8 mode: XeTEX and LuaTEX serve us characters directly, and splitting those into bytes is tricky, encoding dependent, and most likely not useful anyways.

Chapter 9

The l3prg module Control structures

Conditional processing in LaTeX3 is defined as something that performs a series of tests, possibly involving assignments and calling other functions that do not read further ahead in the input stream. After processing the input, a *state* is returned. The states returned are ⟨*true*⟩ and ⟨*false*⟩.

LaTeX3 has two forms of conditional flow processing based on these states. The first form is predicate functions that turn the returned state into a boolean ⟨*true*⟩ or ⟨*false*⟩. For example, the function `\cs_if_free_p:N` checks whether the control sequence given as its argument is free and then returns the boolean ⟨*true*⟩ or ⟨*false*⟩ values to be used in testing with `\if_predicate:w` or in functions to be described below. The second form is the kind of functions choosing a particular argument from the input stream based on the result of the testing as in `\cs_if_free:NTF` which also takes one argument (the `N`) and then executes either `true` or `false` depending on the result.

TeXhackers note: The arguments are executed after exiting the underlying `\if...\fi:` structure.

9.1 Defining a set of conditional functions

`\prg_new_conditional:Npnn`
`\prg_set_conditional:Npnn`
`\prg_gset_conditional:Npnn`
`\prg_new_conditional:Nnn`
`\prg_set_conditional:Nnn`
`\prg_gset_conditional:Nnn`

Updated: 2022-11-01

`\prg_new_conditional:Npnn \⟨name⟩:⟨arg spec⟩ ⟨parameters⟩ {⟨conditions⟩} {⟨code⟩}`
`\prg_new_conditional:Nnn \⟨name⟩:⟨arg spec⟩ {⟨conditions⟩} {⟨code⟩}`

These functions create a family of conditionals using the same {⟨*code*⟩} to perform the test created. Those conditionals are expandable if ⟨*code*⟩ is. The `new` versions check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the `set` versions do no check and perform assignments locally (*cf.* `\cs_set:Npn`). The conditionals created are dependent on the comma-separated list of ⟨*conditions*⟩, which should be one or more of `p`, `T`, `F` and `TF`.

```
\prg_new_protected_conditional:Npnn
\prg_set_protected_conditional:Npnn
\prg_gset_protected_conditional:Npnn
\prg_new_protected_conditional:Nnn
\prg_set_protected_conditional:Nnn
\prg_gset_protected_conditional:Nnn
```

Updated: 2012-02-06

`\prg_new_protected_conditional:Npnn \⟨name⟩:⟨arg spec⟩ ⟨parameters⟩ {⟨conditions⟩} {⟨code⟩}`
`\prg_new_protected_conditional:Nnn \⟨name⟩:⟨arg spec⟩ {⟨conditions⟩} {⟨code⟩}`

These functions create a family of protected conditionals using the same `{`⟨*code*⟩`}` to perform the test created. The ⟨*code*⟩ does not need to be expandable. The `new` version check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the `set` version do not (*cf.* `\cs_set:Npn`). The conditionals created are depended on the comma-separated list of ⟨*conditions*⟩, which should be one or more of `T`, `F` and `TF` (not `p`).

The conditionals are defined by `\prg_new_conditional:Npnn` and friends as:

- `\`⟨*name*⟩`_p:`⟨*arg spec*⟩ — a predicate function which will supply either a logical `true` or logical `false`. This function is intended for use in cases where one or more logical tests are combined to lead to a final outcome. This function cannot be defined for `protected` conditionals.
- `\`⟨*name*⟩`:`⟨*arg spec*⟩`T` — a function with one more argument than the original ⟨*arg spec*⟩ demands. The ⟨*true branch*⟩ code in this additional argument will be left on the input stream only if the test is `true`.
- `\`⟨*name*⟩`:`⟨*arg spec*⟩`F` — a function with one more argument than the original ⟨*arg spec*⟩ demands. The ⟨*false branch*⟩ code in this additional argument will be left on the input stream only if the test is `false`.
- `\`⟨*name*⟩`:`⟨*arg spec*⟩`TF` — a function with two more argument than the original ⟨*arg spec*⟩ demands. The ⟨*true branch*⟩ code in the first additional argument will be left on the input stream if the test is `true`, while the ⟨*false branch*⟩ code in the second argument will be left on the input stream if the test is `false`.

The ⟨*code*⟩ of the test may use ⟨*parameters*⟩ as specified by the second argument to `\prg_set_conditional:Npnn`: this should match the ⟨*argument specification*⟩ but this is not enforced. The `Nnn` versions infer the number of arguments from the argument specification given (*cf.* `\cs_new:Nn`, *etc.*). Within the ⟨*code*⟩, the functions `\prg_return_true:` and `\prg_return_false:` are used to indicate the logical outcomes of the test.

An example can easily clarify matters here:

```
\prg_set_conditional:Npnn \foo_if_bar:NN #1#2 { p , T , TF }
  {
    \if_meaning:w \l_tmpa_tl #1
      \prg_return_true:
    \else:
      \if_meaning:w \l_tmpa_tl #2
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
```

```
    \fi:
  }
```

This defines the function `\foo_if_bar_p:NN`, `\foo_if_bar:NNTF` and `\foo_if_bar:NNT` but not `\foo_if_bar:NNF` (because `F` is missing from the ⟨*conditions*⟩ list). The return statements take care of resolving the remaining `\else:` and `\fi:` before returning the state. There must be a return statement for each branch; failing to do so will result in erroneous output if that branch is executed.

The special case where the code of a conditional ends with `\prg_return_true: \else: \prg_return_false: \fi:` is optimized.

`\prg_new_eq_conditional:NNn`
`\prg_set_eq_conditional:NNn`
`\prg_gset_eq_conditional:NNn`

Updated: 2023-05-26

`\prg_new_eq_conditional:NNn` `\`⟨*name*$_1$⟩`:`⟨*arg spec*$_1$⟩ `\`⟨*name*$_2$⟩`:`⟨*arg spec*$_2$⟩ `{`⟨*conditions*⟩`}`

These functions copy a family of conditionals. The `new` version checks for existing definitions (*cf.* `\cs_new_eq:NN`) whereas the `set` version does not (*cf.* `\cs_set_eq:NN`). The conditionals copied are depended on the comma-separated list of ⟨*conditions*⟩, which should be one or more of `p`, `T`, `F` and `TF`.

`\prg_return_true:` ⋆
`\prg_return_false:` ⋆

`\prg_return_true:`
`\prg_return_false:`

These "return" functions define the logical state of a conditional statement. They appear within the code for a conditional function generated by `\prg_set_conditional:Npnn`, *etc*, to indicate when a true or false branch should be taken. While they may appear multiple times each within the code of such conditionals, the execution of the conditional must result in the expansion of one of these two functions *exactly once*.

The return functions trigger what is internally an `f`-expansion process to complete the evaluation of the conditional. Therefore, after `\prg_return_true:` or `\prg_return_false:` there must be no non-expandable material in the input stream for the remainder of the expansion of the conditional code. This includes other instances of either of these functions.

`\prg_generate_conditional_variant:Nnn`

New: 2017-12-12

`\prg_generate_conditional_variant:Nnn` `\`⟨*name*⟩`:`⟨*arg spec*⟩ `{`⟨*variant argument specifiers*⟩`}` `{`⟨*condition specifiers*⟩`}`

Defines argument-specifier variants of conditionals. This is equivalent to running `\cs_generate_variant:Nn` ⟨*conditional*⟩ `{`⟨*variant argument specifiers*⟩`}` on each ⟨*conditional*⟩ described by the ⟨*condition specifiers*⟩. These base-form ⟨*conditionals*⟩ are obtained from the ⟨*name*⟩ and ⟨*arg spec*⟩ as described for `\prg_new_conditional:Npnn`, and they should be defined.

9.2 The boolean data type

This section describes a boolean data type which is closely connected to conditional processing as sometimes you want to execute some code depending on the value of a switch (*e.g.*, draft/final) and other times you perhaps want to use it as a predicate function in an `\if_predicate:w` test. The problem of the primitive `\if_false:` and

`\if_true:` tokens is that it is not always safe to pass them around as they may interfere with scanning for termination of primitive conditional processing. Therefore, we employ two canonical booleans: `\c_true_bool` or `\c_false_bool`. Besides preventing problems as described above, it also allows us to implement a simple boolean parser supporting the logical operations And, Or, Not, *etc.* which can then be used on both the boolean type and predicate functions.

All conditional `\bool_` functions except assignments are expandable and expect the input to also be fully expandable (which generally means being constructed from predicate functions and booleans, possibly nested).

TeXhackers note: The `bool` data type is not implemented using the `\iffalse`/`\iftrue` primitives, in contrast to `\newif`, *etc.*, in plain TeX, LaTeX 2ε and so on. Programmers should not base use of `bool` switches on any particular expectation of the implementation.

`\bool_new:N`
`\bool_new:c`

`\bool_new:N ⟨boolean⟩`

Creates a new ⟨*boolean*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*boolean*⟩ is initially `false`.

`\bool_const:Nn`
`\bool_const:cn`
New: 2017-11-28

`\bool_const:Nn ⟨boolean⟩ {⟨boolexpr⟩}`

Creates a new constant ⟨*boolean*⟩ or raises an error if the name is already taken. The value of the ⟨*boolean*⟩ is set globally to the result of evaluating the ⟨*boolexpr*⟩.

`\bool_set_false:N`
`\bool_set_false:c`
`\bool_gset_false:N`
`\bool_gset_false:c`

`\bool_set_false:N ⟨boolean⟩`

Sets ⟨*boolean*⟩ logically `false`.

`\bool_set_true:N`
`\bool_set_true:c`
`\bool_gset_true:N`
`\bool_gset_true:c`

`\bool_set_true:N ⟨boolean⟩`

Sets ⟨*boolean*⟩ logically `true`.

`\bool_set_eq:NN`
`\bool_set_eq:(cN|Nc|cc)`
`\bool_gset_eq:NN`
`\bool_gset_eq:(cN|Nc|cc)`

`\bool_set_eq:NN` ⟨*boolean*$_1$⟩ ⟨*boolean*$_2$⟩

Sets ⟨*boolean*$_1$⟩ to the current value of ⟨*boolean*$_2$⟩.

`\bool_set:Nn`
`\bool_set:cn`
`\bool_gset:Nn`
`\bool_gset:cn`
Updated: 2017-07-15

`\bool_set:Nn ⟨boolean⟩ {⟨boolexpr⟩}`

Evaluates the ⟨*boolean expression*⟩ as described for `\bool_if:nTF`, and sets the ⟨*boolean*⟩ variable to the logical truth of this evaluation.

`\bool_set_inverse:N`
`\bool_set_inverse:c`
`\bool_gset_inverse:N`
`\bool_gset_inverse:c`
New: 2018-05-10

`\bool_set_inverse:N ⟨boolean⟩`

Toggles the ⟨*boolean*⟩ from `true` to `false` and conversely: sets it to the inverse of its current value.

`\bool_if_p:N` ⋆
`\bool_if_p:c` ⋆
`\bool_if:NTF` ⋆
`\bool_if:cTF` ⋆

Updated: 2017-07-15

`\bool_if_p:N` ⟨*boolean*⟩
`\bool_if:NTF` ⟨*boolean*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests the current truth of ⟨*boolean*⟩, and continues expansion based on this result.

`\bool_to_str:N` ⋆
`\bool_to_str:c` ⋆
`\bool_to_str:n` ⋆

New: 2021-11-01
Updated: 2023-11-14

`\bool_to_str:N` ⟨*boolean*⟩
`\bool_to_str:n` ⟨*boolean expression*⟩

Expands to the string `true` or `false` depending on the logical truth of the ⟨*boolean*⟩ or ⟨*boolean expression*⟩.

`\bool_show:N`
`\bool_show:c`

New: 2012-02-09
Updated: 2021-04-29

`\bool_show:N` ⟨*boolean*⟩

Displays the logical truth of the ⟨*boolean*⟩ on the terminal.

`\bool_show:n`

New: 2012-02-09
Updated: 2017-07-15

`\bool_show:n` {⟨*boolean expression*⟩}

Displays the logical truth of the ⟨*boolean expression*⟩ on the terminal.

`\bool_log:N`
`\bool_log:c`

New: 2014-08-22
Updated: 2021-04-29

`\bool_log:N` ⟨*boolean*⟩

Writes the logical truth of the ⟨*boolean*⟩ in the log file.

`\bool_log:n`

New: 2014-08-22
Updated: 2017-07-15

`\bool_log:n` {⟨*boolean expression*⟩}

Writes the logical truth of the ⟨*boolean expression*⟩ in the log file.

`\bool_if_exist_p:N` ⋆
`\bool_if_exist_p:c` ⋆
`\bool_if_exist:NTF` ⋆
`\bool_if_exist:cTF` ⋆

New: 2012-03-03

`\bool_if_exist_p:N` ⟨*boolean*⟩
`\bool_if_exist:NTF` ⟨*boolean*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*boolean*⟩ is currently defined. This does not check that the ⟨*boolean*⟩ really is a boolean variable.

9.2.1 Constant and scratch booleans

`\c_true_bool`
`\c_false_bool`

Constants that represent `true` and `false`, respectively. Used to implement predicates.

`\l_tmpa_bool`
`\l_tmpb_bool`

A scratch boolean for local assignment. It is never used by the kernel code, and so is safe for use with any LaTeX3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_bool`
`\g_tmpb_bool`

A scratch boolean for global assignment. It is never used by the kernel code, and so is safe for use with any LaTeX3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

9.3 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean ⟨*true*⟩ or ⟨*false*⟩ values, it seems only fitting that we also provide a parser for ⟨*boolean expressions*⟩.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean ⟨*true*⟩ or ⟨*false*⟩. It supports the logical operations And, Or and Not as the well-known infix operators `&&` and `||` and prefix `!` with their usual precedences (namely, `&&` binds more tightly than `||`). In addition to this, parentheses can be used to isolate sub-expressions. For example,

```
\int_compare_p:n { 1 = 1 } &&
  (
    \int_compare_p:n { 2 = 3 } ||
    \int_compare_p:n { 4 <= 4 } ||
    \str_if_eq_p:nn { abc } { def }
  ) &&
! \int_compare_p:n { 2 = 4 }
```

is a valid boolean expression.

Contrarily to some other programming languages, the operators `&&` and `||` evaluate both operands in all cases, even when the first operand is enough to determine the result. This "eager" evaluation should be contrasted with the "lazy" evaluation of `\bool_lazy_-...` functions.

TeXhackers note: The eager evaluation of boolean expressions is unfortunately necessary in TeX. Indeed, a lazy parser can get confused if `&&` or `||` or parentheses appear as (unbraced) arguments of some predicates. For instance, the innocuous-looking expression below would break (in a lazy parser) if `#1` were a closing parenthesis and `\l_tmpa_bool` were `true`.

```
( \l_tmpa_bool || \token_if_eq_meaning_p:NN X #1 )
```

Minimal (lazy) evaluation can be obtained using the conditionals `\bool_lazy_-all:nTF`, `\bool_lazy_and:nnTF`, `\bool_lazy_any:nTF`, or `\bool_lazy_or:nnTF`, which only evaluate their boolean expression arguments when they are needed to determine the resulting truth value. For example, when evaluating the boolean expression

```
\bool_lazy_and_p:nn
  {
    \bool_lazy_any_p:n
      {
        { \int_compare_p:n { 2 = 3 } }
        { \int_compare_p:n { 4 <= 4 } }
        { \int_compare_p:n { 1 = \error } } % skipped
      }
  }
  { ! \int_compare_p:n { 2 = 4 } }
```

the line marked with `skipped` is not expanded because the result of `\bool_lazy_any_p:n` is known once the second boolean expression is found to be logically `true`. On the other hand, the last line is expanded because its logical value is needed to determine the result of `\bool_lazy_and_p:nn`.

`\bool_if_p:n` ⋆
`\bool_if:nTF` ⋆

Updated: 2017-07-15

`\bool_if_p:n {⟨boolean expression⟩}`
`\bool_if:nTF {⟨boolean expression⟩} {⟨true code⟩} {⟨false code⟩}`

Tests the current truth of ⟨*boolean expression*⟩, and continues expansion based on this result. The ⟨*boolean expression*⟩ should consist of a series of predicates or boolean variables with the logical relationship between these defined using `&&` ("And"), `||` ("Or"), `!` ("Not") and parentheses. The logical Not applies to the next predicate or group.

`\bool_lazy_all_p:n` ⋆
`\bool_lazy_all:nTF` ⋆

New: 2015-11-15
Updated: 2017-07-15

`\bool_lazy_all_p:n { {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} ··· {⟨boolexprN⟩} }`
`\bool_lazy_all:nTF { {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} ··· {⟨boolexprN⟩} } {⟨true code⟩} {⟨false code⟩}`

Implements the "And" operation on the ⟨*boolean expressions*⟩, hence is `true` if all of them are `true` and `false` if any of them is `false`. Contrarily to the infix operator `&&`, only the ⟨*boolean expressions*⟩ which are needed to determine the result of `\bool_lazy_all:nTF` are evaluated. See also `\bool_lazy_and:nnTF` when there are only two ⟨*boolean expressions*⟩.

`\bool_lazy_and_p:nn` ⋆
`\bool_lazy_and:nnTF` ⋆

New: 2015-11-15
Updated: 2017-07-15

`\bool_lazy_and_p:nn {⟨boolexpr₁⟩} {⟨boolexpr₂⟩}`
`\bool_lazy_and:nnTF {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} {⟨true code⟩} {⟨false code⟩}`

Implements the "And" operation between two boolean expressions, hence is `true` if both are `true`. Contrarily to the infix operator `&&`, the ⟨*boolexpr*$_2$⟩ is only evaluated if it is needed to determine the result of `\bool_lazy_and:nnTF`. See also `\bool_lazy_all:nTF` when there are more than two ⟨*boolean expressions*⟩.

`\bool_lazy_any_p:n` ⋆
`\bool_lazy_any:nTF` ⋆

New: 2015-11-15
Updated: 2017-07-15

`\bool_lazy_any_p:n { {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} ··· {⟨boolexprN⟩} }`
`\bool_lazy_any:nTF { {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} ··· {⟨boolexprN⟩} } {⟨true code⟩} {⟨false code⟩}`

Implements the "Or" operation on the ⟨*boolean expressions*⟩, hence is `true` if any of them is `true` and `false` if all of them are `false`. Contrarily to the infix operator `||`, only the ⟨*boolean expressions*⟩ which are needed to determine the result of `\bool_lazy_any:nTF` are evaluated. See also `\bool_lazy_or:nnTF` when there are only two ⟨*boolean expressions*⟩.

`\bool_lazy_or_p:nn` ⋆
`\bool_lazy_or:nnTF` ⋆

New: 2015-11-15
Updated: 2017-07-15

`\bool_lazy_or_p:nn {⟨boolexpr₁⟩} {⟨boolexpr₂⟩}`
`\bool_lazy_or:nnTF {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} {⟨true code⟩} {⟨false code⟩}`

Implements the "Or" operation between two boolean expressions, hence is `true` if either one is `true`. Contrarily to the infix operator `||`, the ⟨*boolexpr*$_2$⟩ is only evaluated if it is needed to determine the result of `\bool_lazy_or:nnTF`. See also `\bool_lazy_any:nTF` when there are more than two ⟨*boolean expressions*⟩.

`\bool_not_p:n` ⋆

Updated: 2017-07-15

`\bool_not_p:n {⟨boolean expression⟩}`

Function version of `!(`⟨*boolean expression*⟩`)` within a boolean expression.

\bool_xor_p:nn ⋆
\bool_xor:nnTF ⋆

New: 2018-05-09

\bool_xor_p:nn {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩}
\bool_xor:nnTF {⟨boolexpr$_1$⟩} {⟨boolexpr$_2$⟩} {⟨true code⟩} {⟨false code⟩}

Implements an "exclusive or" operation between two boolean expressions. There is no infix operation for this logical operation.

9.4 Logical loops

Loops using either boolean expressions or stored boolean values.

\bool_do_until:Nn ☆
\bool_do_until:cn ☆

Updated: 2017-07-15

\bool_do_until:Nn ⟨boolean⟩ {⟨code⟩}

Places the ⟨code⟩ in the input stream for TeX to process, and then checks the logical value of the ⟨boolean⟩. If it is false then the ⟨code⟩ is inserted into the input stream again and the process loops until the ⟨boolean⟩ is true.

\bool_do_while:Nn ☆
\bool_do_while:cn ☆

Updated: 2017-07-15

\bool_do_while:Nn ⟨boolean⟩ {⟨code⟩}

Places the ⟨code⟩ in the input stream for TeX to process, and then checks the logical value of the ⟨boolean⟩. If it is true then the ⟨code⟩ is inserted into the input stream again and the process loops until the ⟨boolean⟩ is false.

\bool_until_do:Nn ☆
\bool_until_do:cn ☆

Updated: 2017-07-15

\bool_until_do:Nn ⟨boolean⟩ {⟨code⟩}

This function first checks the logical value of the ⟨boolean⟩. If it is false the ⟨code⟩ is placed in the input stream and expanded. After the completion of the ⟨code⟩ the truth of the ⟨boolean⟩ is re-evaluated. The process then loops until the ⟨boolean⟩ is true.

\bool_while_do:Nn ☆
\bool_while_do:cn ☆

Updated: 2017-07-15

\bool_while_do:Nn ⟨boolean⟩ {⟨code⟩}

This function first checks the logical value of the ⟨boolean⟩. If it is true the ⟨code⟩ is placed in the input stream and expanded. After the completion of the ⟨code⟩ the truth of the ⟨boolean⟩ is re-evaluated. The process then loops until the ⟨boolean⟩ is false.

\bool_do_until:nn ☆

Updated: 2017-07-15

\bool_do_until:nn {⟨boolean expression⟩} {⟨code⟩}

Places the ⟨code⟩ in the input stream for TeX to process, and then checks the logical value of the ⟨boolean expression⟩ as described for \bool_if:nTF. If it is false then the ⟨code⟩ is inserted into the input stream again and the process loops until the ⟨boolean expression⟩ evaluates to true.

\bool_do_while:nn ☆

Updated: 2017-07-15

\bool_do_while:nn {⟨boolean expression⟩} {⟨code⟩}

Places the ⟨code⟩ in the input stream for TeX to process, and then checks the logical value of the ⟨boolean expression⟩ as described for \bool_if:nTF. If it is true then the ⟨code⟩ is inserted into the input stream again and the process loops until the ⟨boolean expression⟩ evaluates to false.

\bool_until_do:nn ☆

Updated: 2017-07-15

\bool_until_do:nn {⟨boolean expression⟩} {⟨code⟩}

This function first checks the logical value of the ⟨boolean expression⟩ (as described for \bool_if:nTF). If it is false the ⟨code⟩ is placed in the input stream and expanded. After the completion of the ⟨code⟩ the truth of the ⟨boolean expression⟩ is re-evaluated. The process then loops until the ⟨boolean expression⟩ is true.

`\bool_while_do:nn` ☆ `\bool_while_do:nn {⟨boolean expression⟩} {⟨code⟩}`

Updated: 2017-07-15

This function first checks the logical value of the ⟨*boolean expression*⟩ (as described for `\bool_if:nTF`). If it is `true` the ⟨*code*⟩ is placed in the input stream and expanded. After the completion of the ⟨*code*⟩ the truth of the ⟨*boolean expression*⟩ is re-evaluated. The process then loops until the ⟨*boolean expression*⟩ is `false`.

`\bool_case:n` ⋆
`\bool_case:nTF` ⋆

New: 2023-05-03

```
\bool_case:nTF
  {
    {⟨boolexpr case1⟩} {⟨code case1⟩}
    {⟨boolexpr case2⟩} {⟨code case2⟩}
    ...
    {⟨boolexpr casen⟩} {⟨code casen⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

Evaluates in turn each of the ⟨*boolean expression case*⟩s until the first one that evaluates to `true`. The ⟨*code*⟩ associated to this first case is left in the input stream, followed by the ⟨*true code*⟩, and other cases are discarded. If none of the cases match then only the ⟨*false code*⟩ is inserted. The function `\bool_case:n`, which does nothing if there is no match, is also available. For example

```
\bool_case:nF
  {
    { \dim_compare_p:n { \l__mypkg_wd_dim <= 10pt } }
        { Fits }
    { \int_compare_p:n { \l__mypkg_total_int >= 10 } }
        { Many }
    { \l__mypkg_special_bool }
        { Special }
  }
  { No idea! }
```

leaves "`Fits`" or "`Many`" or "`Special`" or "`No idea!`" in the input stream, in a way similar to some other language's "`if` ... `elseif` ... `elseif` ... `else` ...".

9.5 Producing multiple copies

`\prg_replicate:nn` ⋆ `\prg_replicate:nn {⟨integer expression⟩} {⟨tokens⟩}`

Updated: 2011-07-04

Evaluates the ⟨*integer expression*⟩ (which should be zero or positive) and creates the resulting number of copies of the ⟨*tokens*⟩. The function is both expandable and safe for nesting. It yields its result after two expansion steps.

9.6 Detecting TeX's mode

`\mode_if_horizontal_p:` ⋆
`\mode_if_horizontal:TF` ⋆

```
\mode_if_horizontal_p:
\mode_if_horizontal:TF {⟨true code⟩} {⟨false code⟩}
```

Detects if TeX is currently in horizontal mode.

\mode_if_inner_p: ⋆
\mode_if_inner:*TF* ⋆

\mode_if_inner_p:
\mode_if_inner:TF {⟨*true code*⟩} {⟨*false code*⟩}

Detects if TeX is currently in inner mode.

\mode_if_math_p: ⋆
\mode_if_math:*TF* ⋆

Updated: 2011-09-05

\mode_if_math_p:
\mode_if_math:TF {⟨*true code*⟩} {⟨*false code*⟩}

Detects if TeX is currently in maths mode.

\mode_if_vertical_p: ⋆
\mode_if_vertical:*TF* ⋆

\mode_if_vertical_p:
\mode_if_vertical:TF {⟨*true code*⟩} {⟨*false code*⟩}

Detects if TeX is currently in vertical mode.

9.7 Primitive conditionals

\if_predicate:w ⋆

\if_predicate:w ⟨*predicate*⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:

This function takes a predicate function and branches according to the result. (In practice this function would also accept a single boolean variable in place of the ⟨*predicate*⟩ but to make the coding clearer this should be done through \if_bool:N.)

\if_bool:N ⋆

\if_bool:N ⟨*boolean*⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:

This function takes a boolean variable and branches according to the result.

9.8 Nestable recursions and mappings

There are a number of places where recursion or mapping constructs are used in expl3. At a low-level, these typically require insertion of tokens at the end of the content to allow "clean up". To support such mappings in a nestable form, the following functions are provided.

\prg_break_point:Nn ⋆

New: 2018-03-26

\prg_break_point:Nn \⟨*type*⟩_map_break: {⟨*code*⟩}

Used to mark the end of a recursion or mapping: the functions \⟨*type*⟩_map_break: and \⟨*type*⟩_map_break:n use this to break out of the loop (see \prg_map_break:Nn for how to set these up). After the loop ends, the ⟨*code*⟩ is inserted into the input stream. This occurs even if the break functions are *not* applied: \prg_break_point:Nn is functionally-equivalent in these cases to \use_ii:nn.

\prg_map_break:Nn ⋆

New: 2018-03-26

\prg_map_break:Nn \⟨*type*⟩_map_break: {⟨*user code*⟩}
...
\prg_break_point:Nn \⟨*type*⟩_map_break: {⟨*ending code*⟩}

Breaks a recursion in mapping contexts, inserting in the input stream the ⟨*user code*⟩ after the ⟨*ending code*⟩ for the loop. The function breaks loops, inserting their ⟨*ending code*⟩, until reaching a loop with the same ⟨*type*⟩ as its first argument. This \⟨*type*⟩_map_break: argument must be defined; it is simply used as a recognizable marker for the ⟨*type*⟩.

For types with mappings defined in the kernel, \⟨*type*⟩_map_break: and \⟨*type*⟩_map_break:n are defined as \prg_map_break:Nn \⟨*type*⟩_map_break: {} and the same with {} omitted.

9.8.1 Simple mappings

In addition to the more complex mappings above, non-nestable mappings are used in a number of locations and support is provided for these.

\prg_break_point: ⋆

New: 2018-03-27

This copy of \prg_do_nothing: is used to mark the end of a fast short-term recursion: the function \prg_break:n uses this to break out of the loop.

\prg_break: ⋆
\prg_break:n ⋆

New: 2018-03-27

\prg_break:n {⟨*code*⟩} ... \prg_break_point:

Breaks a recursion which has no ⟨*ending code*⟩ and which is not a user-breakable mapping (see for instance implementation of \int_step_function:nnnN), and inserts the ⟨*code*⟩ in the input stream.

9.9 Internal programming functions

\group_align_safe_begin: ⋆
\group_align_safe_end: ⋆

Updated: 2011-08-11

\group_align_safe_begin:
...
\group_align_safe_end:

These functions are used to enclose material in a TeX alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the & token inside \halign. This is necessary to allow grabbing of tokens for testing purposes, as TeX uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as \peek_after:Nw would result in a forbidden comparison of the internal \endtemplate token, yielding a fatal error. Each \group_align_safe_begin: must be matched by a \group_align_safe_end:, although this does not have to occur within the same function.

Chapter 10

The l3sys module System/runtime functions

10.1 The name of the job

\c_sys_jobname_str

New: 2015-09-19
Updated: 2019-10-27

Constant that gets the "job name" assigned when TeX starts.

TeXhackers note: This is the TeX primitive \jobname. For technical reasons, the string here is not of the same internal form as other, but may be manipulated using normal string functions.

10.2 Date and time

\c_sys_minute_int
\c_sys_hour_int
\c_sys_day_int
\c_sys_month_int
\c_sys_year_int

New: 2015-09-22

The date and time at which the current job was started: these are all reported as integers.

TeXhackers note: Whilst the underlying TeX primitives \time, \day, \month, and \year can be altered by the user, this interface to the time and date is intended to be the "real" values.

\c_sys_timestamp_str

New: 2023-08-27

The timestamp for the current job: the format is as described for \file_timestamp:n.

10.3 Engine

\sys_if_engine_luatex_p: ⋆
\sys_if_engine_luatex:TF ⋆
\sys_if_engine_pdftex_p: ⋆
\sys_if_engine_pdftex:TF ⋆
\sys_if_engine_ptex_p: ⋆
\sys_if_engine_ptex:TF ⋆
\sys_if_engine_uptex_p: ⋆
\sys_if_engine_uptex:TF ⋆
\sys_if_engine_xetex_p: ⋆
\sys_if_engine_xetex:TF ⋆

New: 2015-09-07

`\sys_if_engine_pdftex_p:`
`\sys_if_engine_pdftex:TF {⟨true code⟩} {⟨false code⟩}`

Conditionals which allow engine-specific code to be used. The names follow naturally from those of the engine binaries: note that the `(u)ptex` tests are for ε-pTEX and ε-upTEX as expl3 requires the ε-TEX extensions. Each conditional is true for *exactly one* supported engine. In particular, `\sys_if_engine_ptex_p:` is true for ε-pTEX but false for ε-upTEX.

\c_sys_engine_str

New: 2015-09-19

The current engine given as a lower case string: one of `luatex`, `pdftex`, `ptex`, `uptex` or `xetex`.

\c_sys_engine_exec_str

New: 2020-08-20

The name of the standard executable for the current TEX engine given as a lower case string: one of `luatex`, `luahbtex`, `pdftex`, `eptex`, `euptex` or `xetex`.

\c_sys_engine_format_str

New: 2020-08-20

The name of the preloaded format for the current TEX run given as a lower case string: one of `lualatex` (or `dvilualatex`), `pdflatex` (or `latex`), `platex`, `uplatex` or `xelatex` for LATEX, similar names for plain TEX (except pdfTEX in DVI mode yields `etex`), and `cont-en` for ConTEXt (i.e. the `\fmtname`).

\c_sys_engine_version_str

New: 2018-05-02

The version string of the current engine, in the same form as given in the banner issued when running a job. For pdfTEX and LuaTEX this is of the form

⟨*major*⟩.⟨*minor*⟩.⟨*revision*⟩

For XƎTEX, the form is

⟨*major*⟩.⟨*minor*⟩

For pTEX and upTEX, only releases since TEX Live 2018 make the data available, and the form is more complex, as it comprises the pTEX version, the upTEX version and the e-pTEX version.

p⟨*major*⟩.⟨*minor*⟩.⟨*revision*⟩-u⟨*major*⟩.⟨*minor*⟩-⟨*epTeX*⟩

where the `u` part is only present for upTEX.

\sys_timer: ⋆

New: 2021-05-12

`\sys_timer:`

Expands to the current value of the engine's timer clock, a non-negative integer. This function is only defined for engines with timer support. This command measures not just CPU time but real time (including time waiting for user input). The unit are scaled seconds (2^{-16} seconds).

`\sys_if_timer_exist_p:` ⋆
`\sys_if_timer_exist:TF` ⋆

New: 2021-05-12

`\sys_if_timer_exist_p:`
`\sys_if_timer_exist:TF {⟨true code⟩} {⟨false code⟩}`

Tests whether current engine has timer support.

10.4 Output format

`\sys_if_output_dvi_p:` ⋆
`\sys_if_output_dvi:TF` ⋆
`\sys_if_output_pdf_p:` ⋆
`\sys_if_output_pdf:TF` ⋆

New: 2015-09-19

`\sys_if_output_dvi_p:`
`\sys_if_output_dvi:TF {⟨true code⟩} {⟨false code⟩}`

Conditionals which give the current output mode the TeX run is operating in. This is always one of two outcomes, DVI mode or PDF mode. The two sets of conditionals are thus complementary and are both provided to allow the programmer to emphasise the most appropriate case.

`\c_sys_output_str`

New: 2015-09-19

The current output mode given as a lower case string: one of `dvi` or `pdf`.

10.5 Platform

`\sys_if_platform_unix_p:` ⋆
`\sys_if_platform_unix:TF` ⋆
`\sys_if_platform_windows_p:` ⋆
`\sys_if_platform_windows:TF` ⋆

New: 2018-07-27

`\sys_if_platform_unix_p:`
`\sys_if_platform_unix:TF {⟨true code⟩} {⟨false code⟩}`

Conditionals which allow platform-specific code to be used. The names follow the Lua `os.type()` function, *i.e.* all Unix-like systems are `unix` (including Linux and MacOS).

`\c_sys_platform_str`

New: 2018-07-27

The current platform given as a lower case string: one of `unix`, `windows` or `unknown`.

10.6 Random numbers

`\sys_rand_seed:` ⋆

New: 2017-05-27

`\sys_rand_seed:`

Expands to the current value of the engine's random seed, a non-negative integer. In engines without random number support this expands to 0.

`\sys_gset_rand_seed:n` `\sys_gset_rand_seed:n {⟨int expr⟩}`

New: 2017-05-27

Globally sets the seed for the engine's pseudo-random number generator to the ⟨*integer expression*⟩. This random seed affects all `\..._rand` functions (such as `\int_rand:nn` or `\clist_rand_item:n`) as well as other packages relying on the engine's random number generator. In engines without random number support this produces an error.

TEXhackers note: While a 32-bit (signed) integer can be given as a seed, only the absolute value is used and any number beyond 2^{28} is divided by an appropriate power of 2. We recommend using an integer in $[0, 2^{28} - 1]$.

10.7 Access to the shell

`\sys_get_shell:nnN`
`\sys_get_shell:nnNTF`

New: 2019-09-20

`\sys_get_shell:nnN {⟨shell command⟩} {⟨setup⟩} ⟨tl var⟩`
`\sys_get_shell:nnNTF {⟨shell command⟩} {⟨setup⟩} ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}`

Defines ⟨*tl var*⟩ to the text returned by the ⟨*shell command*⟩. The ⟨*shell command*⟩ is converted to a string using `\tl_to_str:n`. Category codes may need to be set appropriately via the ⟨*setup*⟩ argument, which is run just before running the ⟨*shell command*⟩ (in a group). If shell escape is disabled, the ⟨*tl var*⟩ will be set to `\q_no_value` in the non-branching version. Note that quote characters (`"`) *cannot* be used inside the ⟨*shell command*⟩. The `\sys_get_shell:nnNTF` conditional inserts the ⟨*true code*⟩ if the shell is available and no quote is detected, and the ⟨*false code*⟩ otherwise.

Note: It is not possible to tell from TEX if a command is allowed in restricted shell escape. If restricted escape is enabled, the `true` branch is taken: if the command is forbidden at this stage, a low-level TEX error will arise.

`\c_sys_shell_escape_int`

New: 2017-05-27

This variable exposes the internal triple of the shell escape status. The possible values are

0 Shell escape is disabled

1 Unrestricted shell escape is enabled

2 Restricted shell escape is enabled

`\sys_if_shell_p:` ⋆
`\sys_if_shell:TF` ⋆

New: 2017-05-27

`\sys_if_shell_p:`
`\sys_if_shell:TF {⟨true code⟩} {⟨false code⟩}`

Performs a check for whether shell escape is enabled. This returns true if either of restricted or unrestricted shell escape is enabled.

`\sys_if_shell_unrestricted_p:` ⋆
`\sys_if_shell_unrestricted:TF` ⋆

New: 2017-05-27

`\sys_if_shell_unrestricted_p:`
`\sys_if_shell_unrestricted:TF {⟨true code⟩} {⟨false code⟩}`

Performs a check for whether *unrestricted* shell escape is enabled.

\sys_if_shell_restricted_p: ⋆
\sys_if_shell_restricted:TF ⋆

New: 2017-05-27

\sys_if_shell_restricted_p:
\sys_if_shell_restricted:TF {⟨true code⟩} {⟨false code⟩}

Performs a check for whether *restricted* shell escape is enabled. This returns false if unrestricted shell escape is enabled. Unrestricted shell escape is not considered a superset of restricted shell escape in this case. To find whether any shell escape is enabled use \sys_if_shell:TF.

\sys_shell_now:n
\sys_shell_now:e

New: 2017-05-27

\sys_shell_now:n {⟨tokens⟩}

Execute ⟨tokens⟩ through shell escape immediately.

\sys_shell_shipout:n
\sys_shell_shipout:e

New: 2017-05-27

\sys_shell_shipout:n {⟨tokens⟩}

Execute ⟨tokens⟩ through shell escape at shipout.

10.8 System queries

Some queries can be made about the file system, etc., without needing to use unrestricted shell escape. This is carried out using the script l3sys-query, which is documented separately. The wrappers here use this script, if available, to obtain system information that is not directly available within the TeX run. Note that if restricted shell escape is disabled, no results can be obtained.

\sys_get_query:nN
\sys_get_query:nnN
\sys_get_query:nnnN

New: 2024-03-08
Updated: 2024-04-08

\sys_get_query:nN {⟨cmd⟩} ⟨tl var⟩
\sys_get_query:nnN {⟨cmd⟩} {⟨spec⟩} ⟨tl var⟩
\sys_get_query:nnnN {⟨cmd⟩} {⟨options⟩} {⟨spec⟩} ⟨tl var⟩

Sets the ⟨tl var⟩ to the information returned by the l3sys-query ⟨cmd⟩, potentially supplying the ⟨options⟩ and ⟨spec⟩ to the query call. The valid ⟨cmd⟩ names are at present

- pwd Returns the present working directory
- ls Returns a directory listing, using the ⟨spec⟩ to select files and applying the ⟨options⟩ if given

The ⟨spec⟩ is likely to contain the wildcards * or ?, and will automatically be passed to the script without shell expansion. In a glob is needed within the ⟨options⟩, this will need to be protected from shell expansion using ' tokens.

The ⟨spec⟩ and ⟨options⟩, if given, are expanded fully before passing to the underlying script.

Spaces in the output are stored as active tokens, allowing them to be replaced by for example a visible space easily. Other non-letter characters in the ASCII range are set to category code 12. The category codes for characters out of the ASCII range are left unchanged: typically this will mean that with an 8-bit engine, accented values can be typeset directly whilst in Unicode engines, standard category code setup will apply.

If more than one line of text is returned by the ⟨cmd⟩, these will be separated by character 13 (^^M) tokens of category code 12. In most cases, \sys_split_query:nnnN should be preferred when multi-line output is expected.

`\sys_split_query:nN`
`\sys_split_query:nnN`
`\sys_split_query:nnnN`

New: 2024-03-08

`\sys_split_query:nN {⟨cmd⟩} ⟨seq var⟩`
`\sys_split_query:nnN {⟨cmd⟩} {⟨spec⟩} ⟨seq var⟩`
`\sys_split_query:nnnN {⟨cmd⟩} {⟨options⟩} {⟨spec⟩} ⟨seq var⟩`

Works as described for `\sys_split_query:nnnN`, but sets the ⟨*seq var*⟩ to contain one entry for each line returned by `l3sys-query`. This function should therefore be preferred where multi-line return is expected, e.g. for the `ls` command.

10.9 Loading configuration data

`\sys_load_backend:n`

New: 2019-09-12

`\sys_load_backend:n {⟨backend⟩}`

Loads the additional configuration file needed for backend support. If the ⟨*backend*⟩ is empty, the standard backend for the engine in use will be loaded. This command may only be used once.

`\sys_ensure_backend:`

New: 2022-07-29

`\sys_ensure_backend:`

Ensures that a backend has been loaded by calling `\sys_load_backend:n` if required.

`\c_sys_backend_str`

Set to the name of the backend in use by `\sys_load_backend:n` when issued. Possible values are

- `pdftex`
- `luatex`
- `xetex`
- `dvips`
- `dvipdfmx`
- `dvisvgm`

`\sys_load_debug:`

New: 2019-09-12

`\sys_load_debug:`

Load the additional configuration file for debugging support.

10.9.1 Final settings

`\sys_finalise:`

New: 2019-10-06

`\sys_finalise:`

Finalises all system-dependent functionality: required before loading a backend.

Chapter 11

The l3msg module
Messages

Messages need to be passed to the user by modules, either when errors occur or to indicate how the code is proceeding. The l3msg module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by l3msg to create messages divides the process into two distinct parts. Named messages are created in the first part of the process; at this stage, no decision is made about the type of output that the message will produce. The second part of the process is actually producing a message. At this stage a choice of message *class* has to be made, for example `error`, `warning` or `info`.

By separating out the creation and use of messages, several benefits are available. First, the messages can be altered later without needing details of where they are used in the code. This makes it possible to alter the language used, the detail level and so on. Secondly, the output which results from a given message can be altered. This can be done on a message class, module or message name basis. In this way, message behaviour can be altered and messages can be entirely suppressed.

11.1 Creating new messages

All messages have to be created before they can be used. The text of messages is automatically wrapped to the length available in the console. As a result, formatting is only needed where it helps to show meaning. In particular, `\\` may be used to force a new line and `\␣` forces an explicit space. Additionally, `\{`, `\#`, `\}`, `\%` and `\~` can be used to produce the corresponding character.

Messages may be subdivided *by one level* using the `/` character. This is used within the message filtering system to allow for example the LaTeX kernel messages to belong to the module `LaTeX` while still being filterable at a more granular level. Thus for example

```
\msg_new:nnnn { mymodule } { submodule / message } ...
```

will allow to filter out specifically messages from the `submodule`.

Some authors may find the need to include spaces as `~` characters tedious. This can be avoided by locally resetting the category code of `␣`.

```
\char_set_catcode_space:n { `\ }
\msg_new:nnn { foo } { bar }
  {Some message text using '#1' and usual message shorthands \{ \ \ \}.}
\char_set_catcode_ignore:n { `\ }
```

although in general this may be confusing; simply writing the messages using ~ characters is the method favored by the team.

`\msg_new:nnnn`
`\msg_new:nnee`
`\msg_new:nnn`
`\msg_new:nne`

Updated: 2011-08-16

`\msg_new:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Creates a ⟨*message*⟩ for a given ⟨*module*⟩. The message is defined to first give ⟨*text*⟩ and then ⟨*more text*⟩ if the user requests it. If no ⟨*more text*⟩ is available then a standard text is given instead. Within ⟨*text*⟩ and ⟨*more text*⟩ four parameters (#1 to #4) can be used: these will be supplied at the time the message is used. An error is raised if the ⟨*message*⟩ already exists.

`\msg_set:nnnn`
`\msg_set:nnn`

`\msg_set:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Sets up the text for a ⟨*message*⟩ for a given ⟨*module*⟩. The message is defined to first give ⟨*text*⟩ and then ⟨*more text*⟩ if the user requests it. If no ⟨*more text*⟩ is available then a standard text is given instead. Within ⟨*text*⟩ and ⟨*more text*⟩ four parameters (#1 to #4) can be used: these will be supplied at the time the message is used.

`\msg_if_exist_p:nn` ⋆
`\msg_if_exist:nnTF` ⋆

New: 2012-03-03

`\msg_if_exist_p:nn {⟨module⟩} {⟨message⟩}`
`\msg_if_exist:nnTF {⟨module⟩} {⟨message⟩} {⟨true code⟩} {⟨false code⟩}`

Tests whether the ⟨*message*⟩ for the ⟨*module*⟩ is currently defined.

11.2 Customizable information for message modules

`\msg_module_name:n` ⋆

New: 2018-10-10

`\msg_module_name:n {⟨module⟩}`

Expands to the public name of the ⟨*module*⟩ as defined by `\g_msg_module_name_prop` (or otherwise leaves the ⟨*module*⟩ unchanged).

`\msg_module_type:n` ⋆

New: 2018-10-10

`\msg_module_type:n {⟨module⟩}`

Expands to the description which applies to the ⟨*module*⟩, for example a `Package` or `Class`. The information here is defined in `\g_msg_module_type_prop`, and will default to `Package` if an entry is not present.

`\g_msg_module_name_prop`

New: 2018-10-10

Provides a mapping between the module name used for messages, and that for documentation.

`\g_msg_module_type_prop`

New: 2018-10-10

Provides a mapping between the module name used for messages, and that type of module. For example, for LaTeX3 core messages, an empty entry is set here meaning that they are not described using the standard `Package` text.

11.3 Contextual information for messages

\msg_line_context: ☆

\msg_line_context:

Prints the current line number when a message is given, and thus suitable for giving context to messages. The number itself is proceeded by the text `on line`.

\msg_line_number: ⋆

\msg_line_number:

Prints the current line number when a message is given.

\msg_fatal_text:n ⋆

\msg_fatal_text:n {⟨*module*⟩}

Produces the standard text

`Fatal Package` ⟨*module*⟩ `Error`

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included. Any redefinition *must* produce output containing the ⟨*module*⟩ name, and will affect all messages using the expl3 mechanism.

\msg_critical_text:n ⋆

\msg_critical_text:n {⟨*module*⟩}

Produces the standard text

`Critical Package` ⟨*module*⟩ `Error`

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included. Any redefinition *must* produce output containing the ⟨*module*⟩ name, and will affect all messages using the expl3 mechanism.

\msg_error_text:n ⋆

\msg_error_text:n {⟨*module*⟩}

Produces the standard text

`Package` ⟨*module*⟩ `Error`

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included. Any redefinition *must* produce output containing the ⟨*module*⟩ name, and will affect all messages using the expl3 mechanism.

\msg_warning_text:n ⋆

\msg_warning_text:n {⟨*module*⟩}

Produces the standard text

`Package` ⟨*module*⟩ `Warning`

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included. The ⟨*type*⟩ of ⟨*module*⟩ may be adjusted: `Package` is the standard outcome: see `\msg_module_type:n`. Any redefinition *must* produce output containing the ⟨*module*⟩ name, and will affect all messages using the expl3 mechanism.

`\msg_info_text:n` ⋆ `\msg_info_text:n {⟨module⟩}`

Produces the standard text:

`Package ⟨module⟩ Info`

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included. The ⟨*type*⟩ of ⟨*module*⟩ may be adjusted: `Package` is the standard outcome: see `\msg_module_type:n`. Any redefinition *must* produce output containing the ⟨*module*⟩ name, and will affect all messages using the expl3 mechanism.

`\msg_see_documentation_text:n` ⋆ `\msg_see_documentation_text:n {⟨module⟩}`

Updated: 2018-09-30

Produces the standard text

`See the ⟨module⟩ documentation for further information.`

This function can be redefined to alter the language in which the message is given, using `#1` as the name of the ⟨*module*⟩ to be included. The name of the ⟨*module*⟩ is produced using `\msg_module_name:n`.

11.4 Issuing messages

Messages behave differently depending on the message class. In all cases, the message may be issued supplying 0 to 4 arguments. If the number of arguments supplied here does not match the number in the definition of the message, extra arguments are ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments are converted to strings before being added to the message text: the `e`-type variants should be used to expand material. Note that this expansion takes place with the standard definitions in effect, which means that shorthands such as `\~` or `\\` are *not* available; instead one should use `\iow_char:N \~` and `\iow_newline:`, respectively. The following message classes exist:

- `fatal`, ending the TeX run;
- `critical`, ending the file being input;
- `error`, interrupting the TeX run without ending it;
- `warning`, written to terminal and log file, for important messages that may require corrections by the user;
- `note` (less common than `info`) for important information messages written to the terminal and log file;
- `info` for normal information messages written to the log file only;
- `term` and `log` for un-decorated messages written to the terminal and log file, or to the log file only;
- `none` for suppressed messages.

\msg_fatal:nnnnnn
\msg_fatal:nneeee
\msg_fatal:nnnnn
\msg_fatal:(nneee|nnnee)
\msg_fatal:nnnn
\msg_fatal:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_fatal:nnn
\msg_fatal:(nnV|nne)
\msg_fatal:nn

Updated: 2012-08-11

\msg_fatal:nnnnnn {⟨*module*⟩} {⟨*message*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Issues ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. After issuing a fatal error the TEX run halts. No PDF file will be produced in this case (DVI mode runs may produce a truncated DVI file).

\msg_critical:nnnnnn
\msg_critical:nneeee
\msg_critical:nnnnn
\msg_critical:(nneee|nnnee)
\msg_critical:nnnn
\msg_critical:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_critical:nnn
\msg_critical:(nnV|nne)
\msg_critical:nn

Updated: 2012-08-11

\msg_critical:nnnnnn {⟨*module*⟩} {⟨*message*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Issues ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. After issuing a critical error, TEX stops reading the current input file. This may halt the TEX run (if the current file is the main file) or may abort reading a sub-file.

TEXhackers note: The TEX \endinput primitive is used to exit the file. In particular, the rest of the current line remains in the input stream.

\msg_error:nnnnnn
\msg_error:nneeee
\msg_error:nnnnn
\msg_error:(nneee|nnnee)
\msg_error:nnnn
\msg_error:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_error:nnn
\msg_error:(nnV|nne)
\msg_error:nn

Updated: 2012-08-11

\msg_error:nnnnnn {⟨*module*⟩} {⟨*message*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Issues ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The error interrupts processing and issues the text at the terminal. After user input, the run continues.

\msg_warning:nnnnnn
\msg_warning:nneeee
\msg_warning:nnnnn
\msg_warning:(nneee|nnnee)
\msg_warning:nnnn
\msg_warning:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_warning:nnn
\msg_warning:(nnV|nne)
\msg_warning:nn

Updated: 2012-08-11

\msg_warning:nnnnnn {⟨*module*⟩} {⟨*message*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Issues ⟨*module*⟩ warning ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The warning text is added to the log file and the terminal, but the TeX run is not interrupted.

\msg_note:nnnnnn
\msg_note:nneeee
\msg_note:nnnnn
\msg_note:(nneee|nnnee)
\msg_note:nnnn
\msg_note:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_note:nnn
\msg_note:(nnV|nne)
\msg_note:nn
\msg_info:nnnnnn
\msg_info:nneeee
\msg_info:nnnnn
\msg_info:(nneee|nnnee)
\msg_info:nnnn
\msg_info:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_info:nnn
\msg_info:(nnV|nne)
\msg_info:nn

New: 2021-05-18

\msg_note:nnnnnn {⟨*module*⟩} {⟨*message*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}
\msg_info:nnnnnn {⟨*module*⟩} {⟨*message*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Issues ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. For the more common \msg_info:nnnnnn, the information text is added to the log file only, while \msg_note:nnnnnn adds the info text to both the log file and the terminal. The TeX run is not interrupted.

\msg_term:nnnnnn
\msg_term:nneeee
\msg_term:nnnnn
\msg_term:(nneee|nnnee)
\msg_term:nnnn
\msg_term:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_term:nnn
\msg_term:(nnV|nne)
\msg_term:nn
\msg_log:nnnnnn
\msg_log:nneeee
\msg_log:nnnnn
\msg_log:(nneee|nnnee)
\msg_log:nnnn
\msg_log:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_log:nnn
\msg_log:(nnV|nne)
\msg_log:nn

Updated: 2012-08-11

\msg_term:nnnnnn {⟨*module*⟩} {⟨*message*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}
\msg_log:nnnnnn {⟨*module*⟩} {⟨*message*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Issues ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The output is briefer than \msg_info:nnnnnn, omitting for instance the module name. It is added to the log file by \msg_log:nnnnnn while \msg_term:nnnnnn also prints it on the terminal.

\msg_none:nnnnnn
\msg_none:nneeee
\msg_none:nnnnn
\msg_none:(nneee|nnnee)
\msg_none:nnnn
\msg_none:(nnVV|nnVn|nnnV|nnee|nnne)
\msg_none:nnn
\msg_none:(nnV|nne)
\msg_none:nn

Updated: 2012-08-11

\msg_none:nnnnnn {⟨*module*⟩} {⟨*message*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).

11.4.1 Messages for showing material

`\msg_show:nnnnnn`
`\msg_show:nneeee`
`\msg_show:nnnnn`
`\msg_show:(nneee|nnnee)`
`\msg_show:nnnn`
`\msg_show:(nnVV|nnVn|nnnV|nnee|nnne)`
`\msg_show:nnn`
`\msg_show:(nnV|nne)`
`\msg_show:nn`

New: 2017-12-04

`\msg_show:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Issues ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The information text is shown on the terminal and the TeX run is interrupted in a manner similar to `\tl_show:n`. This is used in conjunction with `\msg_show_item:n` and similar functions to print complex variable contents completely. If the formatted text does not contain `>~` at the start of a line, an additional line `>~.` will be put at the end. In addition, a final period is added if not present.

`\msg_show_item:n` ⋆
`\msg_show_item_unbraced:n` ⋆
`\msg_show_item:nn` ⋆
`\msg_show_item_unbraced:nn` ⋆

New: 2017-12-04

`\seq_map_function:NN ⟨seq⟩ \msg_show_item:n`
`\prop_map_function:NN ⟨prop⟩ \msg_show_item:nn`

Used in the text of messages for `\msg_show:nnnnnn` to show or log a list of items or key–value pairs. The output of `\msg_show_item:n` produces a newline, the prefix `>`, two spaces, then the braced string representation of its argument. The two-argument versions separates the key and value using `␣␣=>␣␣`, and the `unbraced` versions don't print the surrounding braces.

These functions are suitable for usage with iterator functions like `\seq_map_function:NN`, `\prop_map_function:NN`, etc. For example, with a sequence `\l_tmpa_seq` containing `a`, `{b}` and `\c`,

```
\seq_map_function:NN \l_tmpa_seq \msg_show_item:n
```

would expand to three lines:

```
>␣␣{a}
>␣␣{{b}}
>␣␣{\c␣}
```

11.4.2 Expandable error messages

In very rare cases it may be necessary to produce errors in an expansion-only context. The functions in this section should only be used if there is no alternative approach using `\msg_error:nnnnnn` or other non-expandable commands from the previous section. Despite having a similar interface as non-expandable messages, expandable errors must be handled internally very differently from normal error messages, as none of the tools to print to the terminal or the log file are expandable. As a result, short-hands such as \{ or \\ do not work, and messages must be very short (with default settings, they are truncated after approximately 50 characters). It is advisable to ensure that the message is understandable even when truncated, by putting the most important information up front. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

\msg_expandable_error:nnnnnn ⋆
\msg_expandable_error:nnffff ⋆
\msg_expandable_error:nnnnn ⋆
\msg_expandable_error:nnfff ⋆
\msg_expandable_error:nnnn ⋆
\msg_expandable_error:nnff ⋆
\msg_expandable_error:nnn ⋆
\msg_expandable_error:nnf ⋆
\msg_expandable_error:nn ⋆

New: 2015-08-06
Updated: 2019-02-28

\msg_expandable_error:nnnnnn {⟨*module*⟩} {⟨*message*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Issues an "Undefined error" message from TEX itself using the undefined control sequence \??? then prints "! ⟨*module*⟩: "⟨*error message*⟩, which should be short. With default settings, anything beyond approximately 60 characters long (or bytes in some engines) is cropped. A leading space might be removed as well.

11.5 Redirecting messages

Each message has a "name", which can be used to alter the behaviour of the message when it is given. Thus we might have

```
\msg_new:nnnn { module } { my-message } { Some~text } { Some~more~text }
```

to define a message, with

```
\msg_error:nn { module } { my-message }
```

when it is used. With no filtering, this raises an error. However, we could alter the behaviour with

```
\msg_redirect_class:nn { error } { warning }
```

to turn all errors into warnings, or with

```
\msg_redirect_module:nnn { module } { error } { warning }
```

to alter only messages from that module, or even

```
\msg_redirect_name:nnn { module } { my-message } { warning }
```

to target just one message. Redirection applies first to individual messages, then to messages from one module and finally to messages of one class. Thus it is possible to select out an individual message for special treatment even if the entire class is already redirected.

Multiple redirections are possible. Redirections can be cancelled by providing an empty argument for the target class. Redirection to a missing class raises an error immediately. Infinite loops are prevented by eliminating the redirection starting from the target of the redirection that caused the loop to appear. Namely, if redirections are requested as $A \to B$, $B \to C$ and $C \to A$ in this order, then the $A \to B$ redirection is cancelled.

`\msg_redirect_class:nn`

Updated: 2012-04-27

`\msg_redirect_class:nn {⟨class one⟩} {⟨class two⟩}`

Changes the behaviour of messages of ⟨*class one*⟩ so that they are processed using the code for those of ⟨*class two*⟩. Each ⟨*class*⟩ can be one of `fatal`, `critical`, `error`, `warning`, `note`, `info`, `term`, `log`, `none`.

`\msg_redirect_module:nnn`

Updated: 2012-04-27

`\msg_redirect_module:nnn {⟨module⟩} {⟨class one⟩} {⟨class two⟩}`

Redirects message of ⟨*class one*⟩ for ⟨*module*⟩ to act as though they were from ⟨*class two*⟩. Messages of ⟨*class one*⟩ from sources other than ⟨*module*⟩ are not affected by this redirection. This function can be used to make some messages "silent" by default. For example, all of the `warning` messages of ⟨*module*⟩ could be turned off with:

```
\msg_redirect_module:nnn { module } { warning } { none }
```

`\msg_redirect_name:nnn`

Updated: 2012-04-27

`\msg_redirect_name:nnn {⟨module⟩} {⟨message⟩} {⟨class⟩}`

Redirects a specific ⟨*message*⟩ from a specific ⟨*module*⟩ to act as a member of ⟨*class*⟩ of messages. No further redirection is performed. This function can be used to make a selected message "silent" without changing global parameters:

```
\msg_redirect_name:nnn { module } { annoying-message } { none }
```

Chapter 12

The l3file module
File and I/O operations

This module provides functions for working with external files. Some of these functions apply to an entire file, and have prefix `\file_...`, while others are used to work with files on a line by line basis and have prefix `\ior_...` (reading) or `\iow_...` (writing).

It is important to remember that when reading external files TeX attempts to locate them using both the operating system path and entries in the TeX file database (most TeX systems use such a database). Thus the "current path" for TeX is somewhat broader than that for other programs.

For functions which expect a ⟨*file name*⟩ argument, this argument may contain both literal items and expandable content, which should on full expansion be the desired file name. Active characters (as declared in `\l_char_active_seq`) are *not* expanded, allowing the direct use of these in file names. Quote tokens (`"`) are not permitted in file names as they are reserved for internal use by some TeX primitives.

Spaces are trimmed at the beginning and end of the file name: this reflects the fact that some file systems do not allow or interact unpredictably with spaces in these positions. When no extension is given, this will trim spaces from the start of the name only.

12.1 Input–output stream management

As TeX engines have a limited number of input and output streams, direct use of the streams by the programmer is not supported in LaTeX3. Instead, an internal pool of streams is maintained, and these are allocated and deallocated as needed by other modules. As a result, the programmer should close streams when they are no longer needed, to release them for other processes.

Note that I/O operations are global: streams should all be declared with global names and treated accordingly.

\ior_new:N
\ior_new:c
\iow_new:N
\iow_new:c

New: 2011-09-26
Updated: 2011-12-27

\ior_new:N ⟨*stream*⟩
\iow_new:N ⟨*stream*⟩

Globally reserves the name of the ⟨*stream*⟩, either for reading or for writing as appropriate. The ⟨*stream*⟩ is not opened until the appropriate \..._open:Nn function is used. Attempting to use a ⟨*stream*⟩ which has not been opened is an error, and the ⟨*stream*⟩ will behave as the corresponding \c_term_....

\ior_open:Nn
\ior_open:cn

Updated: 2012-02-10

\ior_open:Nn ⟨*stream*⟩ {⟨*file name*⟩}

Opens ⟨*file name*⟩ for reading using ⟨*stream*⟩ as the control sequence for file access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*file name*⟩ until an \ior_close:N instruction is given or the TeX run ends. If the file is not found, an error is raised.

\ior_open:Nn*TF*
\ior_open:cn*TF*

New: 2013-01-12

\ior_open:NnTF ⟨*stream*⟩ {⟨*file name*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Opens ⟨*file name*⟩ for reading using ⟨*stream*⟩ as the control sequence for file access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*file name*⟩ until a \ior_close:N instruction is given or the TeX run ends. The ⟨*true code*⟩ is then inserted into the input stream. If the file is not found, no error is raised and the ⟨*false code*⟩ is inserted into the input stream.

\iow_open:Nn
\iow_open:(NV|cn|cV)

Updated: 2012-02-09

\iow_open:Nn ⟨*stream*⟩ {⟨*file name*⟩}

Opens ⟨*file name*⟩ for writing using ⟨*stream*⟩ as the control sequence for file access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*file name*⟩ until a \iow_close:N instruction is given or the TeX run ends. Opening a file for writing clears any existing content in the file (*i.e.* writing is *not* additive).

\ior_shell_open:Nn

New: 2019-05-08

\ior_shell_open:Nn ⟨*stream*⟩ {⟨*shell command*⟩}

Opens the *pseudo*-file created by the output of the ⟨*shell command*⟩ for reading using ⟨*stream*⟩ as the control sequence for access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*shell command*⟩ until a \ior_close:N instruction is given or the TeX run ends. If piped system calls are disabled an error is raised.

For details of handling of the ⟨*shell command*⟩, see \sys_get_shell:nnNTF.

\iow_shell_open:Nn

New: 2023-05-25

\iow_shell_open:Nn ⟨*stream*⟩ {⟨*shell command*⟩}

Opens the *pseudo*-file created by the output of the ⟨*shell command*⟩ for writing using ⟨*stream*⟩ as the control sequence for access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*shell command*⟩ until an \iow_close:N instruction is given or the TeX run ends. If piped system calls are disabled an error is raised.

For details of handling of the ⟨*shell command*⟩, see \sys_get_shell:nnNTF.

\ior_close:N
\ior_close:c
\iow_close:N
\iow_close:c

Updated: 2012-07-31

\ior_close:N ⟨*stream*⟩
\iow_close:N ⟨*stream*⟩

Closes the ⟨*stream*⟩. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.

\ior_show:N
\ior_show:c
\ior_log:N
\ior_log:c
\iow_show:N
\iow_show:c
\iow_log:N
\iow_log:c

New: 2021-05-11

\ior_show:N ⟨*stream*⟩
\ior_log:N ⟨*stream*⟩
\iow_show:N ⟨*stream*⟩
\iow_log:N ⟨*stream*⟩

Display (to the terminal or log file) the file name associated to the (read or write) ⟨*stream*⟩.

\ior_show_list:
\ior_log_list:
\iow_show_list:
\iow_log_list:

New: 2017-06-27

\ior_show_list:
\ior_log_list:
\iow_show_list:
\iow_log_list:

Display (to the terminal or log file) a list of the file names associated with each open (read or write) stream. This is intended for tracking down problems.

12.1.1 Reading from files

Reading from files and reading from the terminal are separate processes in expl3. The functions \ior_get:NN and \ior_str_get:NN, and their branching equivalents, are designed to work with *files*.

\ior_get:NN
\ior_get:NN*TF*

New: 2012-06-24
Updated: 2019-03-23

\ior_get:NN ⟨*stream*⟩ ⟨*tl var*⟩
\ior_get:NNTF ⟨*stream*⟩ ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Function that reads one or more lines (until an equal number of left and right braces are found) from the file input ⟨*stream*⟩ and stores the result locally in the ⟨*token list*⟩ variable. The material read from the ⟨*stream*⟩ is tokenized by TEX according to the category codes and \endlinechar in force when the function is used. Assuming normal settings, any lines which do not end in a comment character % have the line ending converted to a space, so for example input

```
a b  c
```

results in a token list a␣b␣c␣. Any blank line is converted to the token \par. Therefore, blank lines can be skipped by using a test such as

```
\ior_get:NN \l_my_ior \l_tmpa_tl
\tl_set:Nn \l_tmpb_tl { \par }
\tl_if_eq:NNF \l_tmpa_tl \l_tmpb_tl
...
```

Also notice that if multiple lines are read to match braces then the resulting token list can contain \par tokens. In the non-branching version, where the ⟨*stream*⟩ is not open the ⟨*tl var*⟩ is set to \q_no_value.

TEXhackers note: This protected macro is a wrapper around the TEX primitive \read. Regardless of settings, TEX replaces trailing space and tab characters (character codes 32 and 9) in each line by an end-of-line character (character code \endlinechar, omitted if \endlinechar is negative or too large) before turning characters into tokens according to current category codes. With default settings, spaces appearing at the beginning of lines are also ignored.

\ior_str_get:NN
\ior_str_get:NN*TF*

New: 2016-12-04
Updated: 2019-03-23

\ior_str_get:NN ⟨*stream*⟩ ⟨*tl var*⟩
\ior_str_get:NNTF ⟨*stream*⟩ ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Function that reads one line from the file input ⟨*stream*⟩ and stores the result locally in the ⟨*token list*⟩ variable. The material is read from the ⟨*stream*⟩ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). Multiple whitespace characters are retained by this process. It always only reads one line and any blank lines in the input result in the ⟨*tl var*⟩ being empty. Unlike \ior_get:NN, line ends do not receive any special treatment. Thus input

```
a b  c
```

results in a token list a b c with the letters a, b, and c having category code 12. In the non-branching version, where the⟨*stream*⟩ is not open the ⟨*tl var*⟩ is set to \q_-no_value.

TEXhackers note: This protected macro is a wrapper around the ε-TEX primitive \readline. Regardless of settings, TEX removes trailing space and tab characters (character codes 32 and 9). However, the end-line character normally added by this primitive is not included in the result of \ior_str_get:NN.

All mappings are done at the current group level, *i.e.* any local assignments made by the ⟨*function*⟩ or ⟨*code*⟩ discussed below remain in effect after the loop.

`\ior_map_inline:Nn` `\ior_map_inline:Nn` ⟨*stream*⟩ {⟨*inline function*⟩}

New: 2012-02-11

Applies the ⟨*inline function*⟩ to each set of ⟨*lines*⟩ obtained by calling `\ior_get:NN` until reaching the end of the file. TeX ignores any trailing new-line marker from the file it reads. The ⟨*inline function*⟩ should consist of code which receives the ⟨*line*⟩ as `#1`.

`\ior_str_map_inline:Nn` `\ior_str_map_inline:Nn` ⟨*stream*⟩ {⟨*inline function*⟩}

New: 2012-02-11

Applies the ⟨*inline function*⟩ to every ⟨*line*⟩ in the ⟨*stream*⟩. The material is read from the ⟨*stream*⟩ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The ⟨*inline function*⟩ should consist of code which receives the ⟨*line*⟩ as `#1`. Note that TeX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. TeX also ignores any trailing new-line marker from the file it reads.

`\ior_map_variable:NNn` `\ior_map_variable:NNn` ⟨*stream*⟩ ⟨*tl var*⟩ {⟨*code*⟩}

New: 2019-01-13

For each set of ⟨*lines*⟩ obtained by calling `\ior_get:NN` until reaching the end of the file, stores the ⟨*lines*⟩ in the ⟨*tl var*⟩ then applies the ⟨*code*⟩. The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last set of ⟨*lines*⟩, or its original value if the ⟨*stream*⟩ is empty. TeX ignores any trailing new-line marker from the file it reads. This function is typically faster than `\ior_map_inline:Nn`.

`\ior_str_map_variable:NNn` `\ior_str_map_variable:NNn` ⟨*stream*⟩ ⟨*variable*⟩ {⟨*code*⟩}

New: 2019-01-13

For each ⟨*line*⟩ in the ⟨*stream*⟩, stores the ⟨*line*⟩ in the ⟨*variable*⟩ then applies the ⟨*code*⟩. The material is read from the ⟨*stream*⟩ as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last ⟨*line*⟩, or its original value if the ⟨*stream*⟩ is empty. Note that TeX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. TeX also ignores any trailing new-line marker from the file it reads. This function is typically faster than `\ior_str_map_inline:Nn`.

`\ior_map_break:` `\ior_map_break:`

New: 2012-06-29

Used to terminate a `\ior_map_...` function before all lines from the ⟨*stream*⟩ have been processed. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \ior_map_break: }
      {
        % Do something useful
      }
  }
```

Use outside of a `\ior_map_...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\ior_map_break:n` `\ior_map_break:n {`⟨*code*⟩`}`

New: 2012-06-29

Used to terminate a `\ior_map_...` function before all lines in the ⟨*stream*⟩ have been processed, inserting the ⟨*code*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \ior_map_break:n { <code> } }
      {
        % Do something useful
      }
  }
```

Use outside of a `\ior_map_...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the ⟨*code*⟩ is inserted into the input stream. This depends on the design of the mapping function.

`\ior_if_eof_p:N` ⋆ `\ior_if_eof_p:N` ⟨*stream*⟩
`\ior_if_eof:NTF` ⋆ `\ior_if_eof:NTF` ⟨*stream*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Updated: 2012-02-10

Tests if the end of a file ⟨*stream*⟩ has been reached during a reading operation. The test also returns a `true` value if the ⟨*stream*⟩ is not open.

12.1.2 Reading from the terminal

`\ior_get_term:nN`
`\ior_str_get_term:nN`

New: 2019-03-23

`\ior_get_term:nN {⟨prompt⟩} ⟨tl var⟩`

Function that reads one or more lines (until an equal number of left and right braces are found) from the terminal and stores the result locally in the ⟨*token list*⟩ variable. Tokenization occurs as described for `\ior_get:NN` or `\ior_str_get:NN`, respectively. When the ⟨*prompt*⟩ is empty, TeX will wait for input without any other indication: typically the programmer will have provided a suitable text using e.g. `\iow_term:n`. Where the ⟨*prompt*⟩ is given, it will appear in the terminal followed by an `=`, e.g.

```
prompt=
```

12.1.3 Writing to files

`\iow_now:Nn`
`\iow_now:(NV|Ne|cn|cV|ce)`

Updated: 2012-06-05

`\iow_now:Nn ⟨stream⟩ {⟨tokens⟩}`

This function writes ⟨*tokens*⟩ to the specified ⟨*stream*⟩ immediately (*i.e.* the write operation is called on expansion of `\iow_now:Nn`).

`\iow_log:n`
`\iow_log:e`

`\iow_log:n {⟨tokens⟩}`

This function writes the given ⟨*tokens*⟩ to the log (transcript) file immediately: it is a dedicated version of `\iow_now:Nn`.

`\iow_term:n`
`\iow_term:e`

`\iow_term:n {⟨tokens⟩}`

This function writes the given ⟨*tokens*⟩ to the terminal file immediately: it is a dedicated version of `\iow_now:Nn`.

`\iow_shipout:Nn`
`\iow_shipout:(Ne|cn|ce)`

`\iow_shipout:Nn ⟨stream⟩ {⟨tokens⟩}`

This function writes ⟨*tokens*⟩ to the specified ⟨*stream*⟩ when the current page is finalised (*i.e.* at shipout). The `e`-type variants expand the ⟨*tokens*⟩ at the point where the function is used but *not* when the resulting tokens are written to the ⟨*stream*⟩ (*cf.* `\iow_shipout_e:Nn`).

TeXhackers note: When using expl3 with a format other than LaTeX, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

`\iow_shipout_e:Nn`
`\iow_shipout_e:(Ne|cn|ce)`

Updated: 2023-09-17

`\iow_shipout_e:Nn` ⟨*stream*⟩ {⟨*tokens*⟩}

This function writes ⟨*tokens*⟩ to the specified ⟨*stream*⟩ when the current page is finalised (*i.e.* at shipout). The ⟨*tokens*⟩ are expanded at the time of writing in addition to any expansion when the function is used. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).

TeXhackers note: This is a wrapper around the TeX primitive `\write`. When using expl3 with a format other than LaTeX, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

`\iow_char:N` ⋆

`\iow_char:N` \⟨*char*⟩

Inserts ⟨*char*⟩ into the output stream. Useful when trying to write difficult characters such as `%`, `{`, `}`, *etc.* in messages, for example:

```
\iow_now:Ne \g_my_iow { \iow_char:N \{ text \iow_char:N \} }
```

The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

`\iow_newline:` ⋆

`\iow_newline:`

Function to add a new line within the ⟨*tokens*⟩ written to a file. The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

TeXhackers note: When using expl3 with a format other than LaTeX, the character inserted by `\iow_newline:` is not recognized by TeX, which may lead to the insertion of additional unwanted line-breaks. This issue only affects `\iow_shipout:Nn`, `\iow_shipout_e:Nn` and direct uses of primitive operations.

12.1.4 Wrapping lines in output

\iow_wrap:nnnN
\iow_wrap:nenN

New: 2012-06-28
Updated: 2017-12-04

`\iow_wrap:nnnN {⟨text⟩} {⟨run-on text⟩} {⟨set up⟩} ⟨function⟩`

This function wraps the ⟨*text*⟩ to a fixed number of characters per line. At the start of each line which is wrapped, the ⟨*run-on text*⟩ is inserted. The line character count targeted is the value of `\l_iow_line_count_int` minus the number of characters in the ⟨*run-on text*⟩ for all lines except the first, for which the target number of characters is simply `\l_iow_line_count_int` since there is no run-on text. The ⟨*text*⟩ and ⟨*run-on text*⟩ are exhaustively expanded by the function, with the following substitutions:

- `\\` or `\iow_newline:` may be used to force a new line,
- `\␣` may be used to represent a forced space (for example after a control sequence),
- `\#`, `\%`, `\{`, `\}`, `\~` may be used to represent the corresponding character,
- `\iow_wrap_allow_break:` may be used to allow a line-break without inserting a space,
- `\iow_indent:n` may be used to indent a part of the ⟨*text*⟩ (not the ⟨*run-on text*⟩).

Additional functions may be added to the wrapping by using the ⟨*set up*⟩, which is executed before the wrapping takes place: this may include overriding the substitutions listed.

Any expandable material in the ⟨*text*⟩ which is not to be expanded on wrapping should be converted to a string using `\token_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N`, *etc.*

The result of the wrapping operation is passed as a braced argument to the ⟨*function*⟩, which is typically a wrapper around a write operation. The output of `\iow_wrap:nnnN` (*i.e.* the argument passed to the ⟨*function*⟩) consists of characters of category "other" (category code 12), with the exception of spaces which have category "space" (category code 10). This means that the output does *not* expand further when written to a file.

TeXhackers note: Internally, `\iow_wrap:nnnN` carries out an `e`-type expansion on the ⟨*text*⟩ to expand it. This is done in such a way that `\exp_not:N` or `\exp_not:n` *could* be used to prevent expansion of material. However, this is less conceptually clear than conversion to a string, which is therefore the supported method for handling expandable material in the ⟨*text*⟩.

\iow_wrap_allow_break:

New: 2023-04-25

`\iow_wrap_allow_break:`

In the first argument of `\iow_wrap:nnnN` (for instance in messages), inserts a break-point that allows a line break. If no break occurs, this function adds nothing to the output.

\iow_indent:n

New: 2011-09-21

`\iow_indent:n {⟨text⟩}`

In the first argument of `\iow_wrap:nnnN` (for instance in messages), indents ⟨*text*⟩ by four spaces. This function does not cause a line break, and only affects lines which start within the scope of the ⟨*text*⟩. In case the indented ⟨*text*⟩ should appear on separate lines from the surrounding text, use `\\` to force line breaks.

`\l_iow_line_count_int`

New: 2012-06-24

The maximum number of characters in a line to be written by the `\iow_wrap:nnnN` function. This value depends on the TeX system in use: the standard value is 78, which is typically correct for unmodified TeX Live and MiKTeX systems.

12.1.5 Constant input–output streams, and variables

`\g_tmpa_ior`
`\g_tmpb_ior`

New: 2017-12-11

Scratch input stream for global use. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\c_log_iow`
`\c_term_iow`

Constant output streams for writing to the log and to the terminal (plus the log), respectively.

`\g_tmpa_iow`
`\g_tmpb_iow`

New: 2017-12-11

Scratch output stream for global use. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

12.1.6 Primitive conditionals

`\if_eof:w` ⋆

```
\if_eof:w ⟨stream⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Tests if the ⟨*stream*⟩ returns "end of file", which is true for non-existent files. The `\else:` branch is optional.

TeXhackers note: This is the TeX primitive `\ifeof`.

12.2 File operations

12.2.1 Basic file operations

`\g_file_curr_dir_str`
`\g_file_curr_name_str`
`\g_file_curr_ext_str`

New: 2017-06-21

Contain the directory, name and extension of the current file. The directory is empty if the file was loaded without an explicit path (*i.e.* if it is in the TeX search path), and does *not* end in `/` other than the case that it is exactly equal to the root directory. The ⟨*name*⟩ and ⟨*ext*⟩ parts together make up the file name, thus the ⟨*name*⟩ part may be thought of as the "job name" for the current file.

Note that TeX does not provide information on the ⟨*dir*⟩ and ⟨*ext*⟩ part for the main (top level) file and that this file always has empty ⟨*dir*⟩ and ⟨*ext*⟩ components. Also, the ⟨*name*⟩ here will be equal to `\c_sys_jobname_str`, which may be different from the real file name (if set using `--jobname`, for example).

`\l_file_search_path_seq`

New: 2017-06-18
Updated: 2023-06-15

Each entry is the path to a directory which should be searched when seeking a file. Each path can be relative or absolute, and need not include the trailing slash. Spaces need not be quoted.

TeXhackers note: When working as a package in LaTeX 2ε, expl3 will automatically append the current `\input@path` to the set of values from `\l_file_search_path_seq`.

`\file_if_exist_p:n` ⋆
`\file_if_exist_p:V` ⋆
`\file_if_exist:nTF` ⋆
`\file_if_exist:VTF` ⋆

Updated: 2023-09-18

`\file_if_exist_p:n {⟨file name⟩}`
`\file_if_exist:nTF {⟨file name⟩} {⟨true code⟩} {⟨false code⟩}`

Expands the argument of the ⟨*file name*⟩ to give a string, then searches for this string using the current TeX search path and the additional paths controlled by `\l_file_search_path_seq`.

12.2.2 Information about files and file contents

Functions in this section return information about files as expl3 `str` data, *except* that the non-expandable functions set their return *token list* to `\q_no_value` if the file requested is not found. As such, comparison of file names, hashes, sizes, etc., should use `\str_if_eq:nnTF` rather than `\tl_if_eq:nnTF` and so on.

`\file_hex_dump:n` ☆
`\file_hex_dump:V` ☆
`\file_hex_dump:nnn` ☆
`\file_hex_dump:Vnn` ☆

New: 2019-11-19

`\file_hex_dump:n {⟨file name⟩}`
`\file_hex_dump:nnn {⟨file name⟩} {⟨start index⟩} {⟨end index⟩}`

Searches for ⟨*file name*⟩ using the current TeX search path and the additional paths controlled by `\l_file_search_path_seq`. It then expands to leave the hexadecimal dump of the file content in the input stream. The file is read as bytes, which means that in contrast to most TeX behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty. The {⟨*start index*⟩} and {⟨*end index*⟩} values work as described for `\str_range:nnn`.

`\file_get_hex_dump:nN`
`\file_get_hex_dump:VN`
`\file_get_hex_dump:nNTF`
`\file_get_hex_dump:VNTF`
`\file_get_hex_dump:nnnN`
`\file_get_hex_dump:VnnN`
`\file_get_hex_dump:nnnNTF`
`\file_get_hex_dump:VnnNTF`

New: 2019-11-19

`\file_get_hex_dump:nN {⟨file name⟩} ⟨tl var⟩`
`\file_get_hex_dump:nnnN {⟨file name⟩} {⟨start index⟩} {⟨end index⟩} ⟨tl var⟩`

Sets the ⟨*tl var*⟩ to the result of applying `\file_hex_dump:n`/`\file_hex_dump:nnn` to the ⟨*file*⟩. If the file is not found, the ⟨*tl var*⟩ will be set to `\q_no_value`.

\file_mdfive_hash:n ☆
\file_mdfive_hash:V ☆

New: 2019-09-03

\file_mdfive_hash:n {⟨*file name*⟩}

Searches for ⟨*file name*⟩ using the current TEX search path and the additional paths controlled by \l_file_search_path_seq. It then expands to leave the MD5 sum generated from the contents of the file in the input stream. The file is read as bytes, which means that in contrast to most TEX behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty.

\file_get_mdfive_hash:nN
\file_get_mdfive_hash:VN
\file_get_mdfive_hash:nNTF
\file_get_mdfive_hash:VNTF

New: 2017-07-11
Updated: 2019-02-16

\file_get_mdfive_hash:nN {⟨*file name*⟩} ⟨*tl var*⟩

Sets the ⟨*tl var*⟩ to the result of applying \file_mdfive_hash:n to the ⟨*file*⟩. If the file is not found, the ⟨*tl var*⟩ will be set to \q_no_value.

\file_size:n ☆
\file_size:V ☆

New: 2019-09-03

\file_size:n {⟨*file name*⟩}

Searches for ⟨*file name*⟩ using the current TEX search path and the additional paths controlled by \l_file_search_path_seq. It then expands to leave the size of the file in bytes in the input stream. When the file is not found, the result of expansion is empty.

\file_get_size:nN
\file_get_size:VN
\file_get_size:nNTF
\file_get_size:VNTF

New: 2017-07-09
Updated: 2019-02-16

\file_get_size:nN {⟨*file name*⟩} ⟨*tl var*⟩

Sets the ⟨*tl var*⟩ to the result of applying \file_size:n to the ⟨*file*⟩. If the file is not found, the ⟨*tl var*⟩ will be set to \q_no_value.

\file_timestamp:n ☆
\file_timestamp:V ☆

New: 2019-09-03

\file_timestamp:n {⟨*file name*⟩}

Searches for ⟨*file name*⟩ using the current TEX search path and the additional paths controlled by \l_file_search_path_seq. It then expands to leave the modification timestamp of the file in the input stream. The timestamp is of the form D:⟨*year*⟩⟨*month*⟩⟨*day*⟩⟨*hour*⟩⟨*minute*⟩⟨*second*⟩⟨*offset*⟩, where the latter may be Z (UTC) or ⟨*plus-minus*⟩⟨*hours*⟩'⟨*minutes*⟩'. When the file is not found, the result of expansion is empty.

\file_get_timestamp:nN
\file_get_timestamp:VN
\file_get_timestamp:nNTF
\file_get_timestamp:VNTF

New: 2017-07-09
Updated: 2019-02-16

\file_get_timestamp:nN {⟨*file name*⟩} ⟨*tl var*⟩

Sets the ⟨*tl var*⟩ to the result of applying \file_timestamp:n to the ⟨*file*⟩. If the file is not found, the ⟨*tl var*⟩ will be set to \q_no_value.

\file_compare_timestamp_p:nNn ⋆
\file_compare_timestamp_p:(nNV|VNn|VNV) ⋆
\file_compare_timestamp:nNnTF ⋆
\file_compare_timestamp:(nNV|VNn|VNV)TF ⋆

New: 2019-05-13
Updated: 2019-09-20

\file_compare_timestamp_p:nNn {⟨*file-1*⟩} ⟨*relation*⟩ {⟨*file-2*⟩}
\file_compare_timestamp:nNnTF {⟨*file-1*⟩} ⟨*relation*⟩ {⟨*file-2*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Compares the file stamps on the two ⟨*files*⟩ as indicated by the ⟨*relation*⟩, and inserts either the ⟨*true code*⟩ or ⟨*false case*⟩ as required. A file which is not found is treated as older than any file which is found. This allows for example the construct

```
\file_compare_timestamp:nNnT { source-file } > { derived-file }
  {
    % Code to regenerate derived file
  }
```

to work when the derived file is entirely absent. The timestamp of two absent files is regarded as different.

\file_get_full_name:nN
\file_get_full_name:VN
\file_get_full_name:nNTF
\file_get_full_name:VNTF

Updated: 2019-02-16

\file_get_full_name:nN {⟨*file name*⟩} ⟨*tl var*⟩
\file_get_full_name:nNTF {⟨*file name*⟩} ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Searches for ⟨*file name*⟩ in the path as detailed for \file_if_exist:nTF, and if found sets the ⟨*tl var*⟩ the fully-qualified name of the file, *i.e.* the path and file name. This includes an extension .tex when the given ⟨*file name*⟩ has no extension but the file found has that extension. In the non-branching version, the ⟨*tl var*⟩ will be set to \q_no_value in the case that the file does not exist.

\file_full_name:n ☆
\file_full_name:V ☆

New: 2019-09-03

\file_full_name:n {⟨*file name*⟩}

Searches for ⟨*file name*⟩ in the path as detailed for \file_if_exist:nTF, and if found leaves the fully-qualified name of the file, *i.e.* the path and file name, in the input stream. This includes an extension .tex when the given ⟨*file name*⟩ has no extension but the file found has that extension. If the file is not found on the path, the expansion is empty.

\file_parse_full_name:nNNN
\file_parse_full_name:VNNN

New: 2017-06-23
Updated: 2020-06-24

\file_parse_full_name:nNNN {⟨*full name*⟩} ⟨*dir*⟩ ⟨*name*⟩ ⟨*ext*⟩

Parses the ⟨*full name*⟩ and splits it into three parts, each of which is returned by setting the appropriate local string variable:

- The ⟨*dir*⟩: everything up to the last / (path separator) in the ⟨*file path*⟩. As with system PATH variables and related functions, the ⟨*dir*⟩ does *not* include the trailing / unless it points to the root directory. If there is no path (only a file name), ⟨*dir*⟩ is empty.
- The ⟨*name*⟩: everything after the last / up to the last ., where both of those characters are optional. The ⟨*name*⟩ may contain multiple . characters. It is empty if ⟨*full name*⟩ consists only of a directory name.
- The ⟨*ext*⟩: everything after the last . (including the dot). The ⟨*ext*⟩ is empty if there is no . after the last /.

Before parsing, the ⟨*full name*⟩ is expanded until only non-expandable tokens remain, except that active characters are also not expanded. Quotes (") are invalid in file names and are discarded from the input.

\file_parse_full_name:n ⋆
\file_parse_full_name:V ⋆

New: 2020-06-24

\file_parse_full_name:n {⟨*full name*⟩}

Parses the ⟨*full name*⟩ as described for \file_parse_full_name:nNNN, and leaves ⟨*dir*⟩, ⟨*name*⟩, and ⟨*ext*⟩ in the input stream, each inside a pair of braces.

\file_parse_full_name_apply:nN ⋆
\file_parse_full_name_apply:VN ⋆

New: 2020-06-24

\file_parse_full_name_apply:nN {⟨*full name*⟩} ⟨*function*⟩

Parses the ⟨*full name*⟩ as described for \file_parse_full_name:nNNN, and passes ⟨*dir*⟩, ⟨*name*⟩, and ⟨*ext*⟩ as arguments to ⟨*function*⟩, as an n-type argument each, in this order.

12.2.3 Accessing file contents

\file_get:nnN
\file_get:VnN
\file_get:nnN*TF*
\file_get:VnN*TF*

New: 2019-01-16
Updated: 2019-02-16

\file_get:nnN {⟨*file name*⟩} {⟨*setup*⟩} ⟨*tl var*⟩
\file_get:nnNTF {⟨*file name*⟩} {⟨*setup*⟩} ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Defines ⟨*tl var*⟩ to the contents of ⟨*file name*⟩. Category codes may need to be set appropriately via the ⟨*setup*⟩ argument. The non-branching version sets the ⟨*tl var*⟩ to \q_no_value if the file is not found. The branching version runs the ⟨*true code*⟩ after the assignment to ⟨*tl var*⟩ if the file is found, and ⟨*false code*⟩ otherwise. The file content will be tokenized using the current category code régime,

\file_input:n
\file_input:V

Updated: 2017-06-26

\file_input:n {⟨*file name*⟩}

Searches for ⟨*file name*⟩ in the path as detailed for \file_if_exist:nTF, and if found reads in the file as additional LaTeX source. All files read are recorded for information and the file name stack is updated by this function. An error is raised if the file is not found.

\file_input_raw:n ⋆
\file_input_raw:V ⋆

New: 2023-05-18

\file_input_raw:n {⟨*file name*⟩}

Searches for ⟨*file name*⟩ in the path as detailed for \file_if_exist:nTF, and if found reads in the file as additional TeX source. No data concerning the file is tracked. If the file is not found, no action is taken.

TeXhackers note: This function is intended only for contexts where files must be read purely by expansion, for example at the start of a table cell in an \halign.

\file_if_exist_input:n
\file_if_exist_input:V
\file_if_exist_input:nF
\file_if_exist_input:VF

New: 2014-07-02

\file_if_exist_input:n {⟨*file name*⟩}
\file_if_exist_input:nF {⟨*file name*⟩} {⟨*false code*⟩}

Searches for ⟨*file name*⟩ using the current TeX search path and the additional paths included in \l_file_search_path_seq. If found then reads in the file as additional LaTeX source as described for \file_input:n, otherwise inserts the ⟨*false code*⟩. Note that these functions do not raise an error if the file is not found, in contrast to \file_input:n.

`\file_input_stop:` `\file_input_stop:`

New: 2017-07-07

Ends the reading of a file started by `\file_input:n` or similar before the end of the file is reached. Where the file reading is being terminated due to an error, `\msg_critical:nn(nn)` should be preferred.

TeXhackers note: This function must be used on a line on its own: TeX reads files line-by-line and so any additional tokens in the "current" line will still be read.

This is also true if the function is hidden inside another function (which will be the normal case), i.e., all tokens on the same line in the source file are still processed. Putting it on a line by itself in the definition doesn't help as it is the line where it is used that counts!

`\file_show_list:` `\file_show_list:`
`\file_log_list:` `\file_log_list:`

These functions list all files loaded by LaTeX 2_ε commands that populate `\@filelist` or by `\file_input:n`. While `\file_show_list:` displays the list in the terminal, `\file_log_list:` outputs it to the log file only.

Chapter 13

The l3luatex module LuaTeX-specific functions

The LuaTeX engine provides access to the Lua programming language, and with it access to the "internals" of TeX. In order to use this within the framework provided here, a family of functions is available. When used with pdfTeX, pTeX, upTeX or XeTeX these raise an error: use `\sys_if_engine_luatex:T` to avoid this. Details on using Lua with the LuaTeX engine are given in the LuaTeX manual.

13.1 Breaking out to Lua

`\lua_now:n` ⋆
`\lua_now:e` ⋆

New: 2018-06-18

`\lua_now:n {⟨token list⟩}`

The ⟨*token list*⟩ is first tokenized by TeX, which includes converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting ⟨*Lua input*⟩ is passed to the Lua interpreter for processing. Each `\lua_now:n` block is treated by Lua as a separate chunk. The Lua interpreter executes the ⟨*Lua input*⟩ immediately, and in an expandable manner.

TeXhackers note: `\lua_now:e` is a macro wrapper around `\directlua`: when LuaTeX is in use two expansions are required to yield the result of the Lua code.

`\lua_shipout_e:n`
`\lua_shipout:n`

New: 2018-06-18

`\lua_shipout:n {⟨token list⟩}`

The ⟨*token list*⟩ is first tokenized by TeX, which includes converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting ⟨*Lua input*⟩ is passed to the Lua interpreter when the current page is finalised (*i.e.* at shipout). Each `\lua_shipout:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the ⟨*Lua input*⟩ during the page-building routine: no TeX expansion of the ⟨*Lua input*⟩ will occur at this stage.

In the case of the `\lua_shipout_e:n` version the input is fully expanded by TeX in an `e`-type manner during the shipout operation.

TeXhackers note: At a TeX level, the ⟨*Lua input*⟩ is stored as a "whatsit".

\lua_escape:n ⋆
\lua_escape:e ⋆

New: 2015-06-29

\lua_escape:n {⟨*token list*⟩}

Converts the ⟨*token list*⟩ such that it can safely be passed to Lua: embedded backslashes, double and single quotes, and newlines and carriage returns are escaped. This is done by prepending an extra token consisting of a backslash with category code 12, and for the line endings, converting them to \n and \r, respectively.

TEXhackers note: \lua_escape:e is a macro wrapper around \luaescapestring: when LuaTEX is in use two expansions are required to yield the result of the Lua code.

\lua_load_module:n

New: 2022-05-14

\lua_load_module:n {⟨*Lua module name*⟩}

Loads a Lua module into the Lua interpreter.

\lua_now:n passes its {⟨*token list*⟩} argument to the Lua interpreter as a single line, with characters interpreted under the current catcode regime. These two facts mean that \lua_now:n rarely behaves as expected for larger pieces of code. Therefore, package authors should **not** write significant amounts of Lua code in the arguments to \lua_now:n. Instead, it is strongly recommended that they write the majorty of their Lua code in a separate file, and then load it using \lua_load_module:n.

TEXhackers note: This is a wrapper around the Lua call require '⟨*module*⟩'.

13.2 Lua interfaces

As well as interfaces for TEX, there are a small number of Lua functions provided here.

ltx.utils

Most public interfaces provided by the module are stored within the ltx.utils table.

ltx.utils.filedump

⟨*dump*⟩ = ltx.utils.filedump(⟨*file*⟩,⟨*offset*⟩,⟨*length*⟩)

Returns the uppercase hexadecimal representation of the content of the ⟨*file*⟩ read as bytes. If the ⟨*length*⟩ is given, only this part of the file is returned; similarly, one may specify the ⟨*offset*⟩ from the start of the file. If the ⟨*length*⟩ is not given, the entire file is read starting at the ⟨*offset*⟩.

ltx.utils.filemd5sum

⟨*hash*⟩ = ltx.utils.filemd5sum(⟨*file*⟩)

Returns the MD5 sum of the file contents read as bytes; note that the result will depend on the nature of the line endings used in the file, in contrast to normal TEX behaviour. If the ⟨*file*⟩ is not found, nothing is returned with *no error raised.*

ltx.utils.filemoddate

⟨*date*⟩ = ltx.utils.filemoddate(⟨*file*⟩)

Returns the date/time of last modification of the ⟨*file*⟩ in the format

D:⟨*year*⟩⟨*month*⟩⟨*day*⟩⟨*hour*⟩⟨*minute*⟩⟨*second*⟩⟨*offset*⟩

where the latter may be Z (UTC) or ⟨*plus-minus*⟩⟨*hours*⟩'⟨*minutes*⟩'. If the ⟨*file*⟩ is not found, nothing is returned with *no error raised.*

`ltx.utils.filesize` `size = ltx.utils.filesize(`⟨*file*⟩`)`

Returns the size of the ⟨*file*⟩ in bytes. If the ⟨*file*⟩ is not found, nothing is returned with *no error raised*.

Chapter 14

The l3legacy module Interfaces to legacy concepts

There are a small number of TeX or LaTeX 2ε concepts which are not used in expl3 code but which need to be manipulated when working as a LaTeX 2ε package. To allow these to be integrated cleanly into expl3 code, a set of legacy interfaces are provided here.

`\legacy_if_p:n` ★
`\legacy_if:nTF` ★

```
\legacy_if_p:n {⟨name⟩}
\legacy_if:nTF {⟨name⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the LaTeX 2ε/plain TeX conditional (generated by `\newif`) is `true` or `false` and branches accordingly. The ⟨*name*⟩ of the conditional should *omit* the leading `if`.

`\legacy_if_set_true:n`
`\legacy_if_set_false:n`
`\legacy_if_gset_true:n`
`\legacy_if_gset_false:n`

New: 2021-05-10

```
\legacy_if_set_true:n {⟨name⟩}
\legacy_if_set_false:n {⟨name⟩}
```

Sets the LaTeX 2ε/plain TeX conditional `\if`⟨*name*⟩ (generated by `\newif`) to be `true` or `false`.

`\legacy_if_set:nn`
`\legacy_if_gset:nn`

New: 2021-05-10

```
\legacy_if_set:nn {⟨name⟩} {⟨boolexpr⟩}
```

Sets the LaTeX 2ε/plain TeX conditional `\if`⟨*name*⟩ (generated by `\newif`) to the result of evaluating the ⟨*boolean expression*⟩.

Part IV
Data types

Chapter 15

The l3tl module
Token lists

TeX works with tokens, and LaTeX3 therefore provides a number of functions to deal with lists of tokens. Token lists may be present directly in the argument to a function:

```
\foo:n { a collection of \tokens }
```

or may be stored in a so-called "tl var" (⟨*tl var*⟩), which have the suffix `tl`: a token list variable can also be used as the argument to a function, for example

```
\foo:N \l_some_tl
```

In both cases, functions are available to test and manipulate the lists of tokens, and these have the module prefix `tl`. In many cases, functions which can be applied to token list variables are paired with similar functions for application to explicit lists of tokens: the two "views" of a token list are therefore collected together here.

A token list (explicit, or stored in a variable) can be seen either as a list of "items", or a list of "tokens". An item is whatever `\use:n` would grab as its argument: a single non-space token or a brace group, with optional leading explicit space characters (each item is thus itself a token list). A token is either a normal `N` argument, or ␣, `{`, or `}` (assuming normal TeX category codes). Thus for example

```
{ Hello } ~ world
```

contains six items (`Hello`, `w`, `o`, `r`, `l` and `d`), but thirteen tokens (`{`, `H`, `e`, `l`, `l`, `o`, `}`, ␣, `w`, `o`, `r`, `l` and `d`). Functions which act on items are often faster than their analogue acting directly on tokens.

15.1 Creating and initialising token list variables

`\tl_new:N`
`\tl_new:c`

`\tl_new:N` ⟨*tl var*⟩

Creates a new ⟨*tl var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*tl var*⟩ is initially empty.

\tl_const:Nn
\tl_const:(Ne|cn|ce)

\tl_const:Nn ⟨*tl var*⟩ {⟨*tokens*⟩}

Creates a new constant ⟨*tl var*⟩ or raises an error if the name is already taken. The value of the ⟨*tl var*⟩ is set globally to the ⟨*tokens*⟩.

\tl_clear:N
\tl_clear:c
\tl_gclear:N
\tl_gclear:c

\tl_clear:N ⟨*tl var*⟩

Clears all entries from the ⟨*tl var*⟩.

\tl_clear_new:N
\tl_clear_new:c
\tl_gclear_new:N
\tl_gclear_new:c

\tl_clear_new:N ⟨*tl var*⟩

Ensures that the ⟨*tl var*⟩ exists globally by applying \tl_new:N if necessary, then applies \tl_(g)clear:N to leave the ⟨*tl var*⟩ empty.

\tl_set_eq:NN
\tl_set_eq:(cN|Nc|cc)
\tl_gset_eq:NN
\tl_gset_eq:(cN|Nc|cc)

\tl_set_eq:NN ⟨*tl var*$_1$⟩ ⟨*tl var*$_2$⟩

Sets the content of ⟨*tl var*$_1$⟩ equal to that of ⟨*tl var*$_2$⟩.

\tl_concat:NNN
\tl_concat:ccc
\tl_gconcat:NNN
\tl_gconcat:ccc

New: 2012-05-18

\tl_concat:NNN ⟨*tl var*$_1$⟩ ⟨*tl var*$_2$⟩ ⟨*tl var*$_3$⟩

Concatenates the content of ⟨*tl var*$_2$⟩ and ⟨*tl var*$_3$⟩ together and saves the result in ⟨*tl var*$_1$⟩. The ⟨*tl var*$_2$⟩ is placed at the left side of the new token list.

\tl_if_exist_p:N ⋆
\tl_if_exist_p:c ⋆
\tl_if_exist:N*TF* ⋆
\tl_if_exist:c*TF* ⋆

New: 2012-03-03

\tl_if_exist_p:N ⟨*tl var*⟩
\tl_if_exist:NTF ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*tl var*⟩ is currently defined. This does not check that the ⟨*tl var*⟩ really is a token list variable.

15.2 Adding data to token list variables

\tl_set:Nn
\tl_set:(NV|Nv|No|Ne|Nf|cn|cV|cv|co|ce|cf)
\tl_gset:Nn
\tl_gset:(NV|Nv|No|Ne|Nf|cn|cV|cv|co|ce|cf)

\tl_set:Nn ⟨*tl var*⟩ {⟨*tokens*⟩}

Sets ⟨*tl var*⟩ to contain ⟨*tokens*⟩, removing any previous content from the variable.

\tl_put_left:Nn
\tl_put_left:(NV|Nv|Ne|No|cn|cV|cv|ce|co)
\tl_gput_left:Nn
\tl_gput_left:(NV|Nv|Ne|No|cn|cV|cv|ce|co)

\tl_put_left:Nn ⟨*tl var*⟩ {⟨*tokens*⟩}

Appends ⟨*tokens*⟩ to the left side of the current content of ⟨*tl var*⟩.

\tl_put_right:Nn
\tl_put_right:(NV|Nv|Ne|No|cn|cV|cv|ce|co)
\tl_gput_right:Nn
\tl_gput_right:(NV|Nv|Ne|No|cn|cV|cv|ce|co)

\tl_put_right:Nn ⟨*tl var*⟩ {⟨*tokens*⟩}

Appends ⟨*tokens*⟩ to the right side of the current content of ⟨*tl var*⟩.

15.3 Token list conditionals

\tl_if_blank_p:n ⋆
\tl_if_blank_p:(e|V|o) ⋆
\tl_if_blank:n*TF* ⋆
\tl_if_blank:(e|V|o)*TF* ⋆

Updated: 2019-09-04

\tl_if_blank_p:n {⟨*token list*⟩}
\tl_if_blank:nTF {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list*⟩ consists only of blank spaces (*i.e.* contains no item). The test is true if ⟨*token list*⟩ is zero or more explicit space characters (explicit tokens with character code 32 and category code 10), and is false otherwise.

\tl_if_empty_p:N ⋆
\tl_if_empty_p:c ⋆
\tl_if_empty:N*TF* ⋆
\tl_if_empty:c*TF* ⋆

\tl_if_empty_p:N ⟨*tl var*⟩
\tl_if_empty:NTF ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*tl var*⟩ is entirely empty (*i.e.* contains no tokens at all).

\tl_if_empty_p:n ⋆
\tl_if_empty_p:(V|o|e) ⋆
\tl_if_empty:n*TF* ⋆
\tl_if_empty:(V|o|e)*TF* ⋆

New: 2012-05-24
Updated: 2012-06-05

\tl_if_empty_p:n {⟨*token list*⟩}
\tl_if_empty:nTF {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list*⟩ is entirely empty (*i.e.* contains no tokens at all).

\tl_if_eq_p:NN ⋆
\tl_if_eq_p:(Nc|cN|cc) ⋆
\tl_if_eq:NN*TF* ⋆
\tl_if_eq:(Nc|cN|cc)*TF* ⋆

\tl_if_eq_p:NN ⟨*tl var*$_1$⟩ ⟨*tl var*$_2$⟩
\tl_if_eq:NNTF ⟨*tl var*$_1$⟩ ⟨*tl var*$_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Compares the content of ⟨*tl var*$_1$⟩ and ⟨*tl var*$_2$⟩ and is logically true if the two contain the same list of tokens (*i.e.* identical in both the list of characters they contain and the category codes of those characters). Thus for example

```
\tl_set:Nn \l_tmpa_tl { abc }
\tl_set:Ne \l_tmpb_tl { \tl_to_str:n { abc } }
\tl_if_eq:NNTF \l_tmpa_tl \l_tmpb_tl { true } { false }
```

yields false. See also \str_if_eq:nnTF for a comparison that ignores category codes.

\tl_if_eq:Nn*TF*
\tl_if_eq:cn*TF*

New: 2020-07-14

\tl_if_eq:NnTF ⟨*tl var*$_1$⟩ {⟨*token list*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*tl var*$_1$⟩ and the ⟨*token list*$_2$⟩ contain the same list of tokens, both in respect of character codes and category codes. This conditional is not expandable: see \tl_if_eq:NNTF for an expandable version when both token lists are stored in variables, or \str_if_eq:nnTF if category codes are not important.

\tl_if_eq:nnTF
\tl_if_eq:(nV|ne|Vn|en|ee)TF

\tl_if_eq:nnTF {⟨token list₁⟩} {⟨token list₂⟩} {⟨true code⟩} {⟨false code⟩}

Tests if ⟨*token list*$_1$⟩ and ⟨*token list*$_2$⟩ contain the same list of tokens, both in respect of character codes and category codes. This conditional is not expandable: see \tl_if_eq:NNTF for an expandable version when token lists are stored in variables, or \str_if_eq:nnTF if category codes are not important.

\tl_if_in:NnTF
\tl_if_in:(NV|No|cn|cV|co)TF

\tl_if_in:NnTF ⟨tl var⟩ {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨*token list*⟩ is found in the content of the ⟨*tl var*⟩. The ⟨*token list*⟩ cannot contain the tokens {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

\tl_if_in:nnTF
\tl_if_in:(Vn|VV|on|oo|nV|no)TF

\tl_if_in:nnTF {⟨token list₁⟩} {⟨token list₂⟩} {⟨true code⟩} {⟨false code⟩}

Tests if ⟨*token list*$_2$⟩ is found inside ⟨*token list*$_1$⟩. The ⟨*token list*$_2$⟩ cannot contain the tokens {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). The search does *not* enter brace (category code 1/2) groups.

\tl_if_novalue_p:n ⋆
\tl_if_novalue:nTF ⋆

New: 2017-11-14

\tl_if_novalue_p:n {⟨token list⟩}
\tl_if_novalue:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨*token list*⟩ and the special \c_novalue_tl marker contain the same list of tokens, both in respect of character codes and category codes. This means that \exp_args:No \tl_if_novalue:nTF { \c_novalue_tl } is logically true but \tl_if_novalue:nTF { \c_novalue_tl } is logically false. This function is intended to allow construction of flexible document interface structures in which missing optional arguments are detected.

\tl_if_single_p:N ⋆
\tl_if_single_p:c ⋆
\tl_if_single:NTF ⋆
\tl_if_single:cTF ⋆

Updated: 2011-08-13

\tl_if_single_p:N ⟨tl var⟩
\tl_if_single:NTF ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the content of the ⟨*tl var*⟩ consists of a single ⟨*item*⟩, *i.e.* is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to \tl_count:N.

\tl_if_single_p:n ⋆
\tl_if_single:nTF ⋆

Updated: 2011-08-13

\tl_if_single_p:n {⟨token list⟩}
\tl_if_single:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨*token list*⟩ has exactly one ⟨*item*⟩, *i.e.* is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to \tl_count:n.

\tl_if_single_token_p:n ⋆
\tl_if_single_token:nTF ⋆

\tl_if_single_token_p:n {⟨token list⟩}
\tl_if_single_token:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}

Tests if the token list consists of exactly one token, *i.e.* is either a single space character or a single normal token. Token groups ({. . .}) are not single tokens.

15.3.1 Testing the first token

```
\tl_if_head_eq_catcode_p:nN ⋆
\tl_if_head_eq_catcode_p:(VN|eN|oN) ⋆
\tl_if_head_eq_catcode:nNTF ⋆
\tl_if_head_eq_catcode:(VN|eN|oN)TF ⋆
```

Updated: 2012-07-09

```
\tl_if_head_eq_catcode_p:nN {⟨token list⟩} ⟨test token⟩
\tl_if_head_eq_catcode:nNTF {⟨token list⟩} ⟨test token⟩
  {⟨true code⟩} {⟨false code⟩}
```

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ has the same category code as the ⟨*test token*⟩. In the case where the ⟨*token list*⟩ is empty, the test is always `false`.

```
\tl_if_head_eq_charcode_p:nN ⋆
\tl_if_head_eq_charcode_p:(VN|eN|fN) ⋆
\tl_if_head_eq_charcode:nNTF ⋆
\tl_if_head_eq_charcode:(VN|eN|fN)TF ⋆
```

Updated: 2012-07-09

```
\tl_if_head_eq_charcode_p:nN {⟨token list⟩} ⟨test token⟩
\tl_if_head_eq_charcode:nNTF {⟨token list⟩} ⟨test token⟩
  {⟨true code⟩} {⟨false code⟩}
```

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ has the same character code as the ⟨*test token*⟩. In the case where the ⟨*token list*⟩ is empty, the test is always `false`.

```
\tl_if_head_eq_meaning_p:nN ⋆
\tl_if_head_eq_meaning_p:(VN|eN) ⋆
\tl_if_head_eq_meaning:nNTF ⋆
\tl_if_head_eq_meaning:(VN|eN)TF ⋆
```

Updated: 2012-07-09

```
\tl_if_head_eq_meaning_p:nN {⟨token list⟩} ⟨test token⟩
\tl_if_head_eq_meaning:nNTF {⟨token list⟩} ⟨test token⟩
  {⟨true code⟩} {⟨false code⟩}
```

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ has the same meaning as the ⟨*test token*⟩. In the case where ⟨*token list*⟩ is empty, the test is always `false`.

```
\tl_if_head_is_group_p:n ⋆
\tl_if_head_is_group:nTF ⋆
```

New: 2012-07-08

```
\tl_if_head_is_group_p:n {⟨token list⟩}
\tl_if_head_is_group:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ is an explicit begin-group character (with category code 1 and any character code), in other words, if the ⟨*token list*⟩ starts with a brace group. In particular, the test is `false` if the ⟨*token list*⟩ starts with an implicit token such as `\c_group_begin_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

```
\tl_if_head_is_N_type_p:n ⋆
\tl_if_head_is_N_type:nTF ⋆
```

New: 2012-07-08

```
\tl_if_head_is_N_type_p:n {⟨token list⟩}
\tl_if_head_is_N_type:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ is a normal `N`-type argument. In other words, it is neither an explicit space character (explicit token with character code 32 and category code 10) nor an explicit begin-group character (with category code 1 and any character code). An empty argument yields `false`, as it does not have a normal first token. This function is useful to implement actions on token lists on a token by token basis.

`\tl_if_head_is_space_p:n` ⋆
`\tl_if_head_is_space:nTF` ⋆

Updated: 2012-07-08

`\tl_if_head_is_space_p:n {⟨token list⟩}`
`\tl_if_head_is_space:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}`

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ is an explicit space character (explicit token with character code 32 and category code 10). In particular, the test is `false` if the ⟨*token list*⟩ starts with an implicit token such as `\c_space_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

15.4 Working with token lists as a whole

15.4.1 Using token lists

`\tl_to_str:n` ⋆
`\tl_to_str:(o|V|v|e)` ⋆

`\tl_to_str:n {⟨token list⟩}`

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, leaving the resulting character tokens in the input stream. A ⟨*string*⟩ is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space). The base function requires only a single expansion. Its argument *must* be braced.

TeXhackers note: This is the ε-TeX primitive `\detokenize`. Converting a ⟨*token list*⟩ to a ⟨*string*⟩ yields a concatenation of the string representations of every token in the ⟨*token list*⟩. The string representation of a control sequence is

- an escape character, whose character code is given by the internal parameter `\escapechar`, absent if the `\escapechar` is negative or greater than the largest character code;
- the control sequence name, as defined by `\cs_to_str:N`;
- a space, unless the control sequence name is a single character whose category at the time of expansion of `\tl_to_str:n` is not "letter".

The string representation of an explicit character token is that character, doubled in the case of (explicit) macro parameter characters (normally `#`). In particular, the string representation of a token list may depend on the category codes in effect when it is evaluated, and the value of the `\escapechar`: for instance `\tl_to_str:n {\a}` normally produces the three character "backslash", "lower-case a", "space", but it may also produce a single "lower-case a" if the escape character is negative and `a` is currently not a letter.

`\tl_to_str:N` ⋆
`\tl_to_str:c` ⋆

`\tl_to_str:N ⟨tl var⟩`

Converts the content of the ⟨*tl var*⟩ into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This ⟨*string*⟩ is then left in the input stream. For low-level details, see the notes given for `\tl_to_str:n`.

`\tl_use:N` ⋆
`\tl_use:c` ⋆

`\tl_use:N ⟨tl var⟩`

Recovers the content of a ⟨*tl var*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a ⟨*tl var*⟩ directly without an accessor function.

15.4.2 Counting and reversing token lists

\tl_count:n ⋆
\tl_count:(V|v|e|o) ⋆

New: 2012-05-13

\tl_count:n {⟨*token list*⟩}

Counts the number of ⟨*items*⟩ in the ⟨*token list*⟩ and leaves this information in the input stream. Unbraced tokens count as one element as do each token group ({...}). This process ignores any unprotected spaces within the ⟨*token list*⟩. See also \tl_count:N. This function requires three expansions, giving an ⟨*integer denotation*⟩.

\tl_count:N ⋆
\tl_count:c ⋆

New: 2012-05-13

\tl_count:N ⟨*tl var*⟩

Counts the number of ⟨*items*⟩ in the ⟨*tl var*⟩ and leaves this information in the input stream. Unbraced tokens count as one element as do each token group ({...}). This process ignores any unprotected spaces within the ⟨*tl var*⟩. See also \tl_count:n. This function requires three expansions, giving an ⟨*integer denotation*⟩.

\tl_count_tokens:n ⋆

New: 2019-02-25

\tl_count_tokens:n {⟨*token list*⟩}

Counts the number of TeX tokens in the ⟨*token list*⟩ and leaves this information in the input stream. Every token, including spaces and braces, contributes one to the total; thus for instance, the token count of a~{bc} is 6.

\tl_reverse:n ⋆
\tl_reverse:(V|o|f|e) ⋆

Updated: 2012-01-08

\tl_reverse:n {⟨*token list*⟩}

Reverses the order of the ⟨*items*⟩ in the ⟨*token list*⟩, so that ⟨$item_1$⟩⟨$item_2$⟩⟨$item_3$⟩ ...⟨$item_n$⟩ becomes ⟨$item_n$⟩...⟨$item_3$⟩⟨$item_2$⟩⟨$item_1$⟩. This process preserves unprotected space within the ⟨*token list*⟩. Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider \tl_reverse_items:n. See also \tl_reverse:N.

TeXhackers note: The result is returned within \unexpanded, which means that the token list does not expand further when appearing in an e-type or x-type argument expansion.

\tl_reverse:N
\tl_reverse:c
\tl_greverse:N
\tl_greverse:c

Updated: 2012-01-08

\tl_reverse:N ⟨*tl var*⟩

Sets the ⟨*tl var*⟩ to contain the result of reversing the order of its ⟨*items*⟩, so that ⟨$item_1$⟩⟨$item_2$⟩⟨$item_3$⟩ ...⟨$item_n$⟩ becomes ⟨$item_n$⟩...⟨$item_3$⟩⟨$item_2$⟩⟨$item_1$⟩. This process preserves unprotected spaces within the ⟨*tl var*⟩. Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. This is equivalent to a combination of an assignment and \tl_reverse:V. See also \tl_reverse_items:n for improved performance.

\tl_reverse_items:n ⋆

New: 2012-01-08

\tl_reverse_items:n {⟨*token list*⟩}

Reverses the order of the ⟨*items*⟩ in the ⟨*token list*⟩, so that ⟨$item_1$⟩⟨$item_2$⟩⟨$item_3$⟩ ...⟨$item_n$⟩ becomes {⟨$item_n$⟩} ... {⟨$item_3$⟩}{⟨$item_2$⟩}{⟨$item_1$⟩}. This process removes any unprotected space within the ⟨*token list*⟩. Braced token groups are copied without reversing the order of tokens, and keep the outer set of braces. Items which are initially not braced are copied with braces in the result. In cases where preserving spaces is important, consider the slower function \tl_reverse:n.

TeXhackers note: The result is returned within \unexpanded, which means that the token list does not expand further when appearing in an e-type or x-type argument expansion.

\tl_trim_spaces:n ⋆
\tl_trim_spaces:(V|v|e|o) ⋆

New: 2011-07-09
Updated: 2012-06-25

\tl_trim_spaces:n {⟨*token list*⟩}

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the ⟨*token list*⟩ and leaves the result in the input stream.

TEXhackers note: The result is returned within \unexpanded, which means that the token list does not expand further when appearing in an e-type or x-type argument expansion.

\tl_trim_spaces_apply:nN ⋆
\tl_trim_spaces_apply:oN ⋆

New: 2018-04-12

\tl_trim_spaces_apply:nN {⟨*token list*⟩} ⟨*function*⟩

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the ⟨*token list*⟩ and passes the result to the ⟨*function*⟩ as an n-type argument.

\tl_trim_spaces:N
\tl_trim_spaces:c
\tl_gtrim_spaces:N
\tl_gtrim_spaces:c

New: 2011-07-09

\tl_trim_spaces:N ⟨*tl var*⟩

Sets the ⟨*tl var*⟩ to contain the result of removing any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from its contents.

15.4.3 Viewing token lists

\tl_show:N
\tl_show:c

Updated: 2021-04-29

\tl_show:N ⟨*tl var*⟩

Displays the content of the ⟨*tl var*⟩ on the terminal.

TEXhackers note: This is similar to the TEX primitive \show, wrapped to a fixed number of characters per line.

\tl_show:n
\tl_show:e

Updated: 2015-08-07

\tl_show:n {⟨*token list*⟩}

Displays the ⟨*token list*⟩ on the terminal.

TEXhackers note: This is similar to the ε-TEX primitive \showtokens, wrapped to a fixed number of characters per line.

\tl_log:N
\tl_log:c

New: 2014-08-22
Updated: 2021-04-29

\tl_log:N ⟨*tl var*⟩

Writes the content of the ⟨*tl var*⟩ in the log file. See also \tl_show:N which displays the result in the terminal.

\tl_log:n
\tl_log:(e|x)

New: 2014-08-22
Updated: 2015-08-07

\tl_log:n {⟨*token list*⟩}

Writes the ⟨*token list*⟩ in the log file. See also \tl_show:n which displays the result in the terminal.

15.5 Manipulating items in token lists

15.5.1 Mapping over token lists

All mappings are done at the current group level, *i.e.* any local assignments made by the ⟨*function*⟩ or ⟨*code*⟩ discussed below remain in effect after the loop.

`\tl_map_function:NN` ☆
`\tl_map_function:cN` ☆

Updated: 2012-06-29

`\tl_map_function:NN` ⟨*tl var*⟩ ⟨*function*⟩

Applies ⟨*function*⟩ to every ⟨*item*⟩ in the ⟨*tl var*⟩. The ⟨*function*⟩ receives one argument for each iteration. This may be a number of tokens if the ⟨*item*⟩ was stored within braces. Hence the ⟨*function*⟩ should anticipate receiving `n`-type arguments. See also `\tl_map_function:nN`.

`\tl_map_function:nN` ☆

Updated: 2012-06-29

`\tl_map_function:nN {`⟨*token list*⟩`}` ⟨*function*⟩

Applies ⟨*function*⟩ to every ⟨*item*⟩ in the ⟨*token list*⟩, The ⟨*function*⟩ receives one argument for each iteration. This may be a number of tokens if the ⟨*item*⟩ was stored within braces. Hence the ⟨*function*⟩ should anticipate receiving `n`-type arguments. See also `\tl_map_function:NN`.

`\tl_map_inline:Nn`
`\tl_map_inline:cn`

Updated: 2012-06-29

`\tl_map_inline:Nn` ⟨*tl var*⟩ `{`⟨*inline function*⟩`}`

Applies the ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*tl var*⟩. The ⟨*inline function*⟩ should consist of code which receives the ⟨*item*⟩ as `#1`. See also `\tl_map_function:NN`.

`\tl_map_inline:nn`

Updated: 2012-06-29

`\tl_map_inline:nn {`⟨*token list*⟩`} {`⟨*inline function*⟩`}`

Applies the ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*token list*⟩. The ⟨*inline function*⟩ should consist of code which receives the ⟨*item*⟩ as `#1`. See also `\tl_map_function:nN`.

`\tl_map_tokens:Nn` ☆
`\tl_map_tokens:cn` ☆
`\tl_map_tokens:nn` ☆

New: 2019-09-02

`\tl_map_tokens:Nn` ⟨*tl var*⟩ `{`⟨*code*⟩`}`
`\tl_map_tokens:nn {`⟨*token list*⟩`} {`⟨*code*⟩`}`

Analogue of `\tl_map_function:NN` which maps several tokens instead of a single function. The ⟨*code*⟩ receives each ⟨*item*⟩ in the ⟨*tl var*⟩ or in the ⟨*token list*⟩ as a trailing brace group. For instance,

```
\tl_map_tokens:Nn \l_my_tl { \prg_replicate:nn { 2 } }
```

expands to twice each ⟨*item*⟩ in the ⟨*tl var*⟩: for each ⟨*item*⟩ in `\l_my_tl` the function `\prg_replicate:nn` receives `2` and ⟨*item*⟩ as its two arguments. The function `\tl_map_inline:Nn` is typically faster but is not expandable.

`\tl_map_variable:NNn`
`\tl_map_variable:cNn`

Updated: 2012-06-29

`\tl_map_variable:NNn` ⟨*tl var*⟩ ⟨*variable*⟩ `{`⟨*code*⟩`}`

Stores each ⟨*item*⟩ of the ⟨*tl var*⟩ in turn in the (token list) ⟨*variable*⟩ and applies the ⟨*code*⟩. The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last ⟨*item*⟩ in the ⟨*tl var*⟩, or its original value if the ⟨*tl var*⟩ is blank. See also `\tl_map_inline:Nn`.

`\tl_map_variable:nNn` `\tl_map_variable:nNn {⟨token list⟩} ⟨variable⟩ {⟨code⟩}`

Updated: 2012-06-29

Stores each ⟨*item*⟩ of the ⟨*token list*⟩ in turn in the (token list) ⟨*variable*⟩ and applies the ⟨*code*⟩. The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last ⟨*item*⟩ in the ⟨*tl var*⟩, or its original value if the ⟨*tl var*⟩ is blank. See also `\tl_map_inline:nn`.

`\tl_map_break:` ☆ `\tl_map_break:`

Updated: 2012-06-29

Used to terminate a `\tl_map_...` function before all entries in the ⟨*token list*⟩ have been processed. This normally takes place within a conditional statement, for example

```
\tl_map_inline:Nn \l_my_tl
  {
    \str_if_eq:nnT { #1 } { bingo } { \tl_map_break: }
    % Do something useful
  }
```

See also `\tl_map_break:n`. Use outside of a `\tl_map_...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\tl_map_break:n` ☆ `\tl_map_break:n {⟨code⟩}`

Updated: 2012-06-29

Used to terminate a `\tl_map_...` function before all entries in the ⟨*token list*⟩ have been processed, inserting the ⟨*code*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\tl_map_inline:Nn \l_my_tl
  {
    \str_if_eq:nnT { #1 } { bingo }
      { \tl_map_break:n { <code> } }
    % Do something useful
  }
```

Use outside of a `\tl_map_...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the ⟨*code*⟩ is inserted into the input stream. This depends on the design of the mapping function.

15.5.2 Head and tail of token lists

Functions which deal with either only the very first item (balanced text or single normal token) in a token list, or the remaining tokens.

\tl_head:N ⋆
\tl_head:n ⋆
\tl_head:(V|v|f) ⋆

Updated: 2012-09-09

\tl_head:n {⟨*token list*⟩}

Leaves in the input stream the first ⟨*item*⟩ in the ⟨*token list*⟩, discarding the rest of the ⟨*token list*⟩. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example

```
\tl_head:n { abc }
```

and

```
\tl_head:n { ~ abc }
```

both leave a in the input stream. If the "head" is a brace group, rather than a single token, the braces are removed, and so

```
\tl_head:n { ~ { ~ ab } c }
```

yields ␣ab. A blank ⟨*token list*⟩ (see \tl_if_blank:nTF) results in \tl_head:n leaving nothing in the input stream.

TeXhackers note: The result is returned within \exp_not:n, which means that the token list does not expand further when appearing in an e-type or x-type argument expansion.

\tl_head:w ⋆

\tl_head:w ⟨*token list*⟩ { } \q_stop

Leaves in the input stream the first ⟨*item*⟩ in the ⟨*token list*⟩, discarding the rest of the ⟨*token list*⟩. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank ⟨*token list*⟩ (which consists only of space characters) results in a low-level TeX error, which may be avoided by the inclusion of an empty group in the input (as shown), without the need for an explicit test. Alternatively, \tl_if_blank:nF may be used to avoid using the function with a "blank" argument. This function requires only a single expansion, and thus is suitable for use within an o-type expansion. In general, \tl_head:n should be preferred if the number of expansions is not critical.

\tl_tail:N ⋆
\tl_tail:n ⋆
\tl_tail:(V|v|f) ⋆

Updated: 2012-09-01

\tl_tail:n {⟨*token list*⟩}

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first ⟨*item*⟩ in the ⟨*token list*⟩, and leaves the remaining tokens in the input stream. Thus for example

```
\tl_tail:n { a ~ {bc} d }
```

and

```
\tl_tail:n { ~ a ~ {bc} d }
```

both leave ␣{bc}d in the input stream. A blank ⟨*token list*⟩ (see \tl_if_blank:nTF) results in \tl_tail:n leaving nothing in the input stream.

TeXhackers note: The result is returned within \exp_not:n, which means that the token list does not expand further when appearing in an e-type or x-type argument expansion.

If you wish to handle token lists where the first token may be a space, and this needs to be treated as the head/tail, this can be accomplished using \tl_if_head_is_space:nTF, for example

```
\exp_last_unbraced:NNo
  \cs_new:Npn \__mypkg_gobble_space:w \c_space_tl { }
\cs_new:Npn \mypkg_tl_head_keep_space:n #1
  {
    \tl_if_head_is_space:nTF {#1}
      { ~ }
      { \tl_head:n {#1} }
  }
\cs_new:Npn \mypkg_tl_tail_keep_space:n #1
  {
    \tl_if_head_is_space:nTF {#1}
      { \exp_not:o { \__mypkg_gobble_space:w #1 } }
      { \tl_tail:n {#1} }
  }
```

15.5.3 Items and ranges in token lists

\tl_item:nn ⋆
\tl_item:Nn ⋆
\tl_item:cn ⋆

New: 2014-07-17

\tl_item:nn {⟨*token list*⟩} {⟨*integer expression*⟩}

Indexing items in the ⟨*token list*⟩ from 1 on the left, this function evaluates the ⟨*integer expression*⟩ and leaves the appropriate item from the ⟨*token list*⟩ in the input stream. If the ⟨*integer expression*⟩ is negative, indexing occurs from the right of the token list, starting at −1 for the right-most item. If the index is out of bounds, then the function expands to nothing.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨*item*⟩ does not expand further when appearing in an e-type or x-type argument expansion.

\tl_rand_item:N ⋆
\tl_rand_item:c ⋆
\tl_rand_item:n ⋆

New: 2016-12-06

\tl_rand_item:N ⟨*tl var*⟩
\tl_rand_item:n {⟨*token list*⟩}

Selects a pseudo-random item of the ⟨*token list*⟩. If the ⟨*token list*⟩ is blank, the result is empty.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨*item*⟩ does not expand further when appearing in an e-type or x-type argument expansion.

`\tl_range:Nnn` ⋆ `\tl_range:nnn` ⋆

New: 2017-02-17
Updated: 2017-07-15

`\tl_range:Nnn` ⟨*tl var*⟩ {⟨*start index*⟩} {⟨*end index*⟩}
`\tl_range:nnn` {⟨*token list*⟩} {⟨*start index*⟩} {⟨*end index*⟩}

Leaves in the input stream the items from the ⟨*start index*⟩ to the ⟨*end index*⟩ inclusive. Spaces and braces are preserved between the items returned (but never at either end of the list). Here ⟨*start index*⟩ and ⟨*end index*⟩ should be ⟨*integer expressions*⟩. For describing in detail the functions' behavior, let m and n be the start and end index respectively. If either is 0, the result is empty. A positive index means 'start counting from the left end', and a negative index means 'from the right end'. Let l be the count of the token list.

The *actual start point* is determined as $M = m$ if $m > 0$ and as $M = l + m + 1$ if $m < 0$. Similarly the *actual end point* is $N = n$ if $n > 0$ and $N = l + n + 1$ if $n < 0$. If $M > N$, the result is empty. Otherwise it consists of all items from position M to position N inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions s for $s \leq 0$ or $s > l$.

Spaces in between items in the actual range are preserved. Spaces at either end of the token list will be removed anyway (think to the token list being passed to `\tl_trim_spaces:n` to begin with.

Thus, with $l = 7$ as in the examples below, all of the following are equivalent and result in the whole token list

```
\tl_range:nnn { abcd~{e{}}fg } { 1 } { 7 }
\tl_range:nnn { abcd~{e{}}fg } { 1 } { 12 }
\tl_range:nnn { abcd~{e{}}fg } { -7 } { 7 }
\tl_range:nnn { abcd~{e{}}fg } { -12 } { 7 }
```

Here are some more interesting examples. The calls

```
\iow_term:e { \tl_range:nnn { abcd{e{}}fg } { 2 } { 5 } }
\iow_term:e { \tl_range:nnn { abcd{e{}}fg } { 2 } { -3 } }
\iow_term:e { \tl_range:nnn { abcd{e{}}fg } { -6 } { 5 } }
\iow_term:e { \tl_range:nnn { abcd{e{}}fg } { -6 } { -3 } }
```

are all equivalent and will print `bcd{e{}}` on the terminal; similarly

```
\iow_term:e { \tl_range:nnn { abcd~{e{}}fg } { 2 } { 5 } }
\iow_term:e { \tl_range:nnn { abcd~{e{}}fg } { 2 } { -3 } }
\iow_term:e { \tl_range:nnn { abcd~{e{}}fg } { -6 } { 5 } }
\iow_term:e { \tl_range:nnn { abcd~{e{}}fg } { -6 } { -3 } }
```

are all equivalent and will print `bcd {e{}}` on the terminal (note the space in the middle). To the contrary,

```
\tl_range:nnn { abcd~{e{}}f } { 2 } { 4 }
```

will discard the space after 'd'.

If we want to get the items from, say, the third to the last in a token list `<tl>`, the call is `\tl_range:nnn { <tl> } { 3 } { -1 }`. Similarly, for discarding the last item, we can do `\tl_range:nnn { <tl> } { 1 } { -2 }`.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*item*⟩ does not expand further when appearing in an `e`-type or `x`-type argument expansion.

15.5.4 Sorting token lists

\tl_sort:Nn
\tl_sort:cn
\tl_gsort:Nn
\tl_gsort:cn

New: 2017-02-06

\tl_sort:Nn ⟨*tl var*⟩ {⟨*comparison code*⟩}

Sorts the items in the ⟨*tl var*⟩ according to the ⟨*comparison code*⟩, and assigns the result to ⟨*tl var*⟩. The details of sorting comparison are described in Section 6.1.

\tl_sort:nN ⋆

New: 2017-02-06

\tl_sort:nN {⟨*token list*⟩} ⟨*conditional*⟩

Sorts the items in the ⟨*token list*⟩, using the ⟨*conditional*⟩ to compare items, and leaves the result in the input stream. The ⟨*conditional*⟩ should have signature `:nnTF`, and return `true` if the two items being compared should be left in the same order, and `false` if the items should be swapped. The details of sorting comparison are described in Section 6.1.

TEXhackers note: The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an `e`-type or `x`-type argument expansion.

15.6 Manipulating tokens in token lists

15.6.1 Replacing tokens

Within token lists, replacement takes place at the top level: there is no recursion into brace groups (more precisely, within a group defined by a category code 1/2 pair).

\tl_replace_once:Nnn
\tl_replace_once:(NVn|NnV|Nen|Nne|Nee|cnn|cVn|cnV|cen|cne|cee)
\tl_greplace_once:Nnn
\tl_greplace_once:(NVn|NnV|Nen|Nne|Nee|cnn|cVn|cnV|cen|cne|cee)

Updated: 2011-08-11

\tl_replace_once:Nnn ⟨*tl var*⟩ {⟨*old tokens*⟩} {⟨*new tokens*⟩}

Replaces the first (leftmost) occurrence of ⟨*old tokens*⟩ in the ⟨*tl var*⟩ with ⟨*new tokens*⟩. ⟨*Old tokens*⟩ cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

`\tl_replace_all:Nnn`
`\tl_replace_all:(NVn|NnV|Nen|Nne|Nee|cnn|cVn|cnV|cen|cne|cee)`
`\tl_greplace_all:Nnn`
`\tl_greplace_all:(NVn|NnV|Nen|Nne|Nee|cnn|cVn|cnV|cen|cne|cee)`

Updated: 2011-08-11

`\tl_replace_all:Nnn` ⟨*tl var*⟩ {⟨*old tokens*⟩} {⟨*new tokens*⟩}

Replaces all occurrences of ⟨*old tokens*⟩ in the ⟨*tl var*⟩ with ⟨*new tokens*⟩. ⟨*Old tokens*⟩ cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern ⟨*old tokens*⟩ may remain after the replacement (see `\tl_remove_all:Nn` for an example).

`\tl_remove_once:Nn`
`\tl_remove_once:(NV|Ne|cn|cV|ce)`
`\tl_gremove_once:Nn`
`\tl_gremove_once:(NV|Ne|cn|cV|ce)`

Updated: 2011-08-11

`\tl_remove_once:Nn` ⟨*tl var*⟩ {⟨*tokens*⟩}

Removes the first (leftmost) occurrence of ⟨*tokens*⟩ from the ⟨*tl var*⟩. The ⟨*tokens*⟩ cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

`\tl_remove_all:Nn`
`\tl_remove_all:(NV|Ne|cn|cV|ce)`
`\tl_gremove_all:Nn`
`\tl_gremove_all:(NV|Ne|cn|cV|ce)`

Updated: 2011-08-11

`\tl_remove_all:Nn` ⟨*tl var*⟩ {⟨*tokens*⟩}

Removes all occurrences of ⟨*tokens*⟩ from the ⟨*tl var*⟩. The ⟨*tokens*⟩ cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern ⟨*tokens*⟩ may remain after the removal, for instance,

```
\tl_set:Nn \l_tmpa_tl {abbccd} \tl_remove_all:Nn \l_tmpa_tl {bc}
```

results in `\l_tmpa_tl` containing `abcd`.

15.6.2 Reassigning category codes

These functions allow the rescanning of tokens: re-apply TeX's tokenization process to apply category codes different from those in force when the tokens were absorbed. Whilst this functionality is supported, it is often preferable to find alternative approaches to achieving outcomes rather than rescanning tokens (for example construction of token lists token-by-token with intervening category code changes or using `\char_generate:nn`).

\tl_set_rescan:Nnn
\tl_set_rescan:(NnV|Nne|Nno|cnn|cnV|cne|cno)
\tl_gset_rescan:Nnn
\tl_gset_rescan:(NnV|Nne|Nno|cnn|cnV|cne|cno)

Updated: 2015-08-11

\tl_set_rescan:Nnn ⟨*tl var*⟩ {⟨*setup*⟩} {⟨*tokens*⟩}

Sets ⟨*tl var*⟩ to contain ⟨*tokens*⟩, applying the category code régime specified in the ⟨*setup*⟩ before carrying out the assignment. (Category codes applied to tokens not explicitly covered by the ⟨*setup*⟩ are those in force at the point of use of \tl_set_rescan:Nnn.) This allows the ⟨*tl var*⟩ to contain material with category codes other than those that apply when ⟨*tokens*⟩ are absorbed. The ⟨*setup*⟩ is run within a group and may contain any valid input, although only changes in category codes, such as uses of \cctab_select:N, are relevant. See also \tl_rescan:nn.

TEXhackers note: The ⟨*tokens*⟩ are first turned into a string (using \tl_to_str:n). If the string contains one or more characters with character code \newlinechar (set equal to \endlinechar unless that is equal to 32, before the user ⟨*setup*⟩), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

\tl_rescan:nn
\tl_rescan:nV

Updated: 2015-08-11

\tl_rescan:nn {⟨*setup*⟩} {⟨*tokens*⟩}

Rescans ⟨*tokens*⟩ applying the category code régime specified in the ⟨*setup*⟩, and leaves the resulting tokens in the input stream. (Category codes applied to tokens not explicitly covered by the ⟨*setup*⟩ are those in force at the point of use of \tl_rescan:nn.) The ⟨*setup*⟩ is run within a group and may contain any valid input, although only changes in category codes, such as uses of \cctab_select:N, are relevant. See also \tl_set_rescan:Nnn, which is more robust than using \tl_set:Nn in the ⟨*tokens*⟩ argument of \tl_rescan:nn.

TEXhackers note: The ⟨*tokens*⟩ are first turned into a string (using \tl_to_str:n). If the string contains one or more characters with character code \newlinechar (set equal to \endlinechar unless that is equal to 32, before the user ⟨*setup*⟩), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

Contrarily to the \scantokens ε-TEX primitive, \tl_rescan:nn tokenizes the whole string in the same category code regime rather than one token at a time, so that directives such as \verb that rely on changing category codes will not function properly.

15.7 Constant token lists

\c_empty_tl

Constant that is always empty.

`\c_novalue_tl`
New: 2017-11-14

A marker for the absence of an argument. This constant `tl` can safely be typeset (*cf.* `\q_nil`), with the result being `-NoValue-`. It is important to note that `\c_novalue_tl` is constructed such that it will *not* match the simple text input `-NoValue-`, *i.e.* that

```
\tl_if_eq:NnTF \c_novalue_tl { -NoValue- }
```

is logically `false`. The `\c_novalue_tl` marker is intended for use in creating document-level interfaces, where it serves as an indicator that an (optional) argument was omitted. In particular, it is distinct from a simple empty `tl`.

`\c_space_tl`

An explicit space character contained in a token list (compare this with `\c_space_token`). For use where an explicit space is required.

15.8 Scratch token lists

`\l_tmpa_tl`
`\l_tmpb_tl`

Scratch token lists for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_tl`
`\g_tmpb_tl`

Scratch token lists for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Chapter 16

The l3tl-build module Piecewise tl constructions

16.1 Constructing ⟨*tl var*⟩ by accumulation

When creating a ⟨*tl var*⟩ by accumulation of many tokens, the performance available using a combination of `\tl_set:Nn` and `\tl_put_right:Nn` or similar begins to become an issue. To address this, a set of functions are available to "build" a ⟨*tl var*⟩. The performance of this approach is much more efficient than the standard `\tl_put_right:Nn`, but the constructed token list cannot be accessed during construction other than by methods provided in this section.

Whilst the exact performance difference is dependent on the size of each added block of tokens and the total number of blocks, in general, the `\tl_build_(g)put...` functions will out-perform the basic `\tl_(g)put...` equivalent if more than 100 non-empty addition operations occur. See https://github.com/latex3/latex3/issues/1393#issuecomment-1880164756 for a more detailed analysis.

`\tl_build_begin:N`
`\tl_build_gbegin:N`

New: 2018-04-01

`\tl_build_begin:N` ⟨*tl var*⟩

Clears the ⟨*tl var*⟩ and sets it up to support other `\tl_build_...` functions. Until `\tl_build_end:N` ⟨*tl var*⟩ or `\tl_build_gend:N` ⟨*tl var*⟩ is called, applying any function from l3tl other than `\tl_build_...` will lead to incorrect results. The `begin` and `gbegin` functions must be used for local and global ⟨*tl var*⟩ respectively.

`\tl_build_put_left:Nn`
`\tl_build_put_left:Ne`
`\tl_build_gput_left:Nn`
`\tl_build_gput_left:Ne`
`\tl_build_put_right:Nn`
`\tl_build_put_right:Ne`
`\tl_build_gput_right:Nn`
`\tl_build_gput_right:Ne`

New: 2018-04-01

`\tl_build_put_left:Nn` ⟨*tl var*⟩ {⟨*tokens*⟩}
`\tl_build_put_right:Nn` ⟨*tl var*⟩ {⟨*tokens*⟩}

Adds ⟨*tokens*⟩ to the left or right side of the current contents of ⟨*tl var*⟩. The ⟨*tl var*⟩ must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `put` and `gput` functions must be used for local and global ⟨*tl var*⟩ respectively. The `right` functions are about twice faster than the `left` functions.

`\tl_build_end:N`
`\tl_build_gend:N`

New: 2018-04-01

`\tl_build_end:N` ⟨*tl var*⟩

Gets the contents of ⟨*tl var*⟩ and stores that into the ⟨*tl var*⟩ using `\tl_set:Nn` or `\tl_gset:Nn`. The ⟨*tl var*⟩ must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `end` and `gend` functions must be used for local and global ⟨*tl var*⟩ respectively. These functions completely remove the setup code that enabled ⟨*tl var*⟩ to be used for other `\tl_build_...` functions. After the action of `end`/`gend`, the ⟨*tl var*⟩ may be manipulated using standard `tl` functions.

`\tl_build_get_intermediate:NN`

New: 2023-12-14

`\tl_build_get_intermediate:NN` ⟨*tl var*$_1$⟩ ⟨*tl var*$_2$⟩

Stores the contents of the ⟨*tl var*$_1$⟩ in the ⟨*tl var*$_2$⟩. The ⟨*tl var*$_1$⟩ must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The ⟨*tl var*$_2$⟩ is a "normal" token list variable, assigned locally using `\tl_set:Nn`.

Chapter 17

The l3str module Strings

TEX associates each character with a category code: as such, there is no concept of a "string" as commonly understood in many other programming languages. However, there are places where we wish to manipulate token lists while in some sense "ignoring" category codes: this is done by treating token lists as strings in a TEX sense.

A TEX string (and thus an expl3 string) is a series of characters which have category code 12 ("other") with the exception of space characters which have category code 10 ("space"). Thus at a technical level, a TEX string is a token list with the appropriate category codes. In this documentation, these are simply referred to as strings.

String variables are simply specialised token lists, but by convention should be named with the suffix `...str`. Such variables should contain characters with category code 12 (other), except spaces, which have category code 10 (blank space). All the functions in this module which accept a token list argument first convert it to a string using `\tl_to_str:n` for internal processing, and do not treat a token list or the corresponding string representation differently.

As a string is a subset of the more general token list, it is sometimes unclear when one should be used over the other. Use a string variable for data that isn't primarily intended for typesetting and for which a level of protection from unwanted expansion is suitable. This data type simplifies comparison of variables since there are no concerns about expansion of their contents.

The functions `\cs_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N` and `\token_to_str:N` (and variants) generate strings from the appropriate input: these are documented in l3basics, l3tl and l3token, respectively.

Most expandable functions in this module come in three flavours:

- `\str_...:N`, which expect a token list or string variable as their argument;
- `\str_...:n`, taking any token list (or string) as an argument;
- `\str_..._ignore_spaces:n`, which ignores any space encountered during the operation: these functions are typically faster than those which take care of escaping spaces appropriately.

17.1 Creating and initialising string variables

\str_new:N
\str_new:c
New: 2015-09-18

\str_new:N ⟨*str var*⟩

Creates a new ⟨*str var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*str var*⟩ is initially empty.

\str_const:Nn
\str_const:(NV|Ne|cn|cV|ce)
New: 2015-09-18
Updated: 2018-07-28

\str_const:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Creates a new constant ⟨*str var*⟩ or raises an error if the name is already taken. The value of the ⟨*str var*⟩ is set globally to the ⟨*token list*⟩, converted to a string.

\str_clear:N
\str_clear:c
\str_gclear:N
\str_gclear:c
New: 2015-09-18

\str_clear:N ⟨*str var*⟩

Clears the content of the ⟨*str var*⟩.

\str_clear_new:N
\str_clear_new:c
\str_gclear_new:N
\str_gclear_new:c
New: 2015-09-18

\str_clear_new:N ⟨*str var*⟩

Ensures that the ⟨*str var*⟩ exists globally by applying \str_new:N if necessary, then applies \str_(g)clear:N to leave the ⟨*str var*⟩ empty.

\str_set_eq:NN
\str_set_eq:(cN|Nc|cc)
\str_gset_eq:NN
\str_gset_eq:(cN|Nc|cc)
New: 2015-09-18

\str_set_eq:NN ⟨*str var*$_1$⟩ ⟨*str var*$_2$⟩

Sets the content of ⟨*str var*$_1$⟩ equal to that of ⟨*str var*$_2$⟩.

\str_concat:NNN
\str_concat:ccc
\str_gconcat:NNN
\str_gconcat:ccc
New: 2017-10-08

\str_concat:NNN ⟨*str var*$_1$⟩ ⟨*str var*$_2$⟩ ⟨*str var*$_3$⟩

Concatenates the content of ⟨*str var*$_2$⟩ and ⟨*str var*$_3$⟩ together and saves the result in ⟨*str var*$_1$⟩. The ⟨*str var*$_2$⟩ is placed at the left side of the new string variable. The ⟨*str var*$_2$⟩ and ⟨*str var*$_3$⟩ must indeed be strings, as this function does not convert their contents to a string.

\str_if_exist_p:N ⋆
\str_if_exist_p:c ⋆
\str_if_exist:N*TF* ⋆
\str_if_exist:c*TF* ⋆
New: 2015-09-18

\str_if_exist_p:N ⟨*str var*⟩
\str_if_exist:NTF ⟨*str var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*str var*⟩ is currently defined. This does not check that the ⟨*str var*⟩ really is a string.

17.2 Adding data to string variables

\str_set:Nn
\str_set:(NV|Ne|cn|cV|ce)
\str_gset:Nn
\str_gset:(NV|Ne|cn|cV|ce)

New: 2015-09-18
Updated: 2018-07-28

\str_set:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and stores the result in ⟨*str var*⟩.

\str_put_left:Nn
\str_put_left:(NV|Ne|cn|cV|ce)
\str_gput_left:Nn
\str_gput_left:(NV|Ne|cn|cV|ce)

New: 2015-09-18
Updated: 2018-07-28

\str_put_left:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and prepends the result to ⟨*str var*⟩. The current contents of the ⟨*str var*⟩ are not automatically converted to a string.

\str_put_right:Nn
\str_put_right:(NV|Ne|cn|cV|Ne)
\str_gput_right:Nn
\str_gput_right:(NV|Ne|cn|cV|ce)

New: 2015-09-18
Updated: 2018-07-28

\str_put_right:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and appends the result to ⟨*str var*⟩. The current contents of the ⟨*str var*⟩ are not automatically converted to a string.

17.3 String conditionals

\str_if_empty_p:N ⋆
\str_if_empty_p:c ⋆
\str_if_empty:N*TF* ⋆
\str_if_empty:c*TF* ⋆
\str_if_empty_p:n ⋆
\str_if_empty:n*TF* ⋆

New: 2015-09-18
Updated: 2022-03-21

\str_if_empty_p:N ⟨*str var*⟩
\str_if_empty:NTF ⟨*str var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*string variable*⟩ is entirely empty (*i.e.* contains no characters at all).

\str_if_eq_p:NN ⋆
\str_if_eq_p:(Nc|cN|cc) ⋆
\str_if_eq:NN*TF* ⋆
\str_if_eq:(Nc|cN|cc)*TF* ⋆

New: 2015-09-18

\str_if_eq_p:NN ⟨*str var*$_1$⟩ ⟨*str var*$_2$⟩
\str_if_eq:NNTF ⟨*str var*$_1$⟩ ⟨*str var*$_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Compares the content of two ⟨*str variables*⟩ and is logically `true` if the two contain the same characters in the same order. See `\tl_if_eq:NNTF` to compare tokens (including their category codes) rather than characters.

```
\str_if_eq_p:nn                              ⋆ \str_if_eq_p:nn {⟨tl1⟩} {⟨tl2⟩}
\str_if_eq_p:(Vn|on|no|nV|VV|vn|nv|ee)       ⋆ \str_if_eq:nnTF {⟨tl1⟩} {⟨tl2⟩} {⟨true code⟩} {⟨false code⟩}
\str_if_eq:nnTF                              ⋆
\str_if_eq:(Vn|on|no|nV|VV|vn|nv|ee)TF       ⋆
```

Updated: 2018-06-18

Compares the two ⟨*token lists*⟩ on a character by character basis (namely after converting them to strings), and is `true` if the two ⟨*strings*⟩ contain the same characters in the same order. Thus for example

```
\str_if_eq_p:no { abc } { \tl_to_str:n { abc } }
```

is logically `true`. See `\tl_if_eq:nnTF` to compare tokens (including their category codes) rather than characters.

```
\str_if_in:NnTF   \str_if_in:NnTF ⟨str var⟩ {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
\str_if_in:cnTF
```

New: 2017-10-08

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ and tests if that ⟨*string*⟩ is found in the content of the ⟨*str var*⟩.

```
\str_if_in:nnTF   \str_if_in:nnTF {⟨tl1⟩} {⟨tl2⟩} {⟨true code⟩} {⟨false code⟩}
```

New: 2017-10-08

Converts both ⟨*token lists*⟩ to ⟨*strings*⟩ and tests whether ⟨*string*$_2$⟩ is found inside ⟨*string*$_1$⟩.

```
\str_case:nn                       ⋆ \str_case:nnTF {⟨test string⟩}
\str_case:(Vn|on|en|nV|nv)         ⋆   {
\str_case:nnTF                     ⋆     {⟨string case1⟩} {⟨code case1⟩}
\str_case:(Vn|on|en|nV|nv)TF       ⋆     {⟨string case2⟩} {⟨code case2⟩}
\str_case:Nn                       ⋆     ...
\str_case:NnTF                     ⋆     {⟨string casen⟩} {⟨code casen⟩}
                                       }
                                       {⟨true code⟩}
                                       {⟨false code⟩}
```

New: 2013-07-24
Updated: 2022-03-21

Compares the ⟨*test string*⟩ in turn with each of the ⟨*string case*⟩s until a match is found (all token lists are converted to strings). If the two are equal (as described for `\str_if_eq:nnTF`) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\str_case:nn`, which does nothing if there is no match, is also available.

This set of functions performs no expansion on each ⟨*string case*⟩ argument, so any variable in there will be compared as a string. If expansion is needed in the ⟨*string case*⟩s, then `\str_case_e:nn(TF)` should be used instead.

\str_case_e:nn ⋆ \str_case_e:en ⋆ \str_case_e:nnTF ⋆ \str_case_e:enTF ⋆ New: 2018-06-19	

```
\str_case_e:nnTF {⟨test string⟩}
  {
    {⟨string case_1⟩} {⟨code case_1⟩}
    {⟨string case_2⟩} {⟨code case_2⟩}
    ...
    {⟨string case_n⟩} {⟨code case_n⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

Compares the full expansion of the ⟨*test string*⟩ in turn with the full expansion of the ⟨*string case*⟩s (all token lists are converted to strings). If the two full expansions are equal (as described for `\str_if_eq:eeTF`) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\str_case_e:nn`, which does nothing if there is no match, is also available. In `\str_case_e:nn(TF)`, the ⟨*test string*⟩ is expanded in each comparison, and must always yield the same result: for example, random numbers must not be used within this string.

\str_compare_p:nNn ⋆ \str_compare_p:eNe ⋆ \str_compare:nNnTF ⋆ \str_compare:eNeTF ⋆ New: 2021-05-17	

```
\str_compare_p:nNn {⟨tl_1⟩} ⟨relation⟩ {⟨tl_2⟩}
\str_compare:nNnTF {⟨tl_1⟩} ⟨relation⟩ {⟨tl_2⟩} {⟨true code⟩} {⟨false code⟩}
```

Compares the two ⟨*token lists*⟩ on a character by character basis (namely after converting them to strings) in a lexicographic order according to the character codes of the characters. The ⟨*relation*⟩ can be `<`, `=`, or `>` and the test is `true` under the following conditions:

- for `<`, if the first string is earlier than the second in lexicographic order;
- for `=`, if the two strings have exactly the same characters;
- for `>`, if the first string is later than the second in lexicographic order.

Thus for example the following is logically `true`:

```
\str_compare_p:nNn { ab } < { abc }
```

TEXhackers note: This is a wrapper around the TEX primitive `\(pdf)strcmp`. It is meant for programming and not for sorting textual contents, as it simply considers character codes and not more elaborate considerations of grapheme clusters, locale, etc.

17.4 Mapping over strings

All mappings are done at the current group level, *i.e.* any local assignments made by the ⟨*function*⟩ or ⟨*code*⟩ discussed below remain in effect after the loop.

\str_map_function:nN ☆ \str_map_function:NN ☆ \str_map_function:cN ☆ New: 2017-11-14	

```
\str_map_function:nN {⟨token list⟩} ⟨function⟩
\str_map_function:NN ⟨str var⟩ ⟨function⟩
```

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ then applies ⟨*function*⟩ to every ⟨*character*⟩ in the ⟨*string*⟩ including spaces.

\str_map_inline:nn
\str_map_inline:Nn
\str_map_inline:cn

New: 2017-11-14

\str_map_inline:nn {⟨*token list*⟩} {⟨*inline function*⟩}
\str_map_inline:Nn ⟨*str var*⟩ {⟨*inline function*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ then applies the ⟨*inline function*⟩ to every ⟨*character*⟩ in the ⟨*str var*⟩ including spaces. The ⟨*inline function*⟩ should consist of code which receives the ⟨*character*⟩ as #1.

\str_map_tokens:nn ☆
\str_map_tokens:Nn ☆
\str_map_tokens:cn ☆

New: 2021-05-05

\str_map_tokens:nn {⟨*token list*⟩} {⟨*code*⟩}
\str_map_tokens:Nn ⟨*str var*⟩ {⟨*code*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ then applies ⟨*code*⟩ to every ⟨*character*⟩ in the ⟨*string*⟩ including spaces. The ⟨*code*⟩ receives each character as a trailing brace group. This is equivalent to \str_map_function:nN if the ⟨*code*⟩ consists of a single function.

\str_map_variable:nNn
\str_map_variable:NNn
\str_map_variable:cNn

New: 2017-11-14

\str_map_variable:nNn {⟨*token list*⟩} ⟨*variable*⟩ {⟨*code*⟩}
\str_map_variable:NNn ⟨*str var*⟩ ⟨*variable*⟩ {⟨*code*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ then stores each ⟨*character*⟩ in the ⟨*string*⟩ (including spaces) in turn in the (string or token list) ⟨*variable*⟩ and applies the ⟨*code*⟩. The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last ⟨*character*⟩ in the ⟨*string*⟩, or its original value if the ⟨*string*⟩ is empty. See also \str_map_inline:Nn.

\str_map_break: ☆

New: 2017-10-08

\str_map_break:

Used to terminate a \str_map_... function before all characters in the ⟨*string*⟩ have been processed. This normally takes place within a conditional statement, for example

```
\str_map_inline:Nn \l_my_str
  {
    \str_if_eq:nnT { #1 } { bingo } { \str_map_break: }
    % Do something useful
  }
```

See also \str_map_break:n. Use outside of a \str_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before continuing with the code that follows the loop. This depends on the design of the mapping function.

\str_map_break:n ☆ \str_map_break:n {⟨*code*⟩}

New: 2017-10-08

Used to terminate a \str_map_... function before all characters in the ⟨*string*⟩ have been processed, inserting the ⟨*code*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\str_map_inline:Nn \l_my_str
  {
    \str_if_eq:nnT { #1 } { bingo }
      { \str_map_break:n { <code> } }
    % Do something useful
  }
```

Use outside of a \str_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the ⟨*code*⟩ is inserted into the input stream. This depends on the design of the mapping function.

17.5 Working with the content of strings

\str_use:N ⋆
\str_use:c ⋆

New: 2015-09-18

\str_use:N ⟨*str var*⟩

Recovers the content of a ⟨*str var*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a ⟨*str*⟩ directly without an accessor function.

\str_count:N ⋆
\str_count:c ⋆
\str_count:n ⋆
\str_count_ignore_spaces:n ⋆

New: 2015-09-18

\str_count:n {⟨*token list*⟩}

Leaves in the input stream the number of characters in the string representation of ⟨*token list*⟩, as an integer denotation. The functions differ in their treatment of spaces. In the case of \str_count:N and \str_count:n, all characters including spaces are counted. The \str_count_ignore_spaces:n function leaves the number of non-space characters in the input stream.

\str_count_spaces:N ⋆
\str_count_spaces:c ⋆
\str_count_spaces:n ⋆

New: 2015-09-18

\str_count_spaces:n {⟨*token list*⟩}

Leaves in the input stream the number of space characters in the string representation of ⟨*token list*⟩, as an integer denotation. Of course, this function has no _ignore_spaces variant.

\str_head:N ⋆
\str_head:c ⋆
\str_head:n ⋆
\str_head_ignore_spaces:n ⋆

New: 2015-09-18

\str_head:n {⟨*token list*⟩}

Converts the ⟨*token list*⟩ into a ⟨*string*⟩. The first character in the ⟨*string*⟩ is then left in the input stream, with category code "other". The functions differ if the first character is a space: \str_head:N and \str_head:n return a space token with category code 10 (blank space), while the \str_head_ignore_spaces:n function ignores this space character and leaves the first non-space character in the input stream. If the ⟨*string*⟩ is empty (or only contains spaces in the case of the _ignore_spaces function), then nothing is left on the input stream.

\str_tail:N ⋆
\str_tail:c ⋆
\str_tail:n ⋆
\str_tail_ignore_spaces:n ⋆

New: 2015-09-18

\str_tail:n {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, removes the first character, and leaves the remaining characters (if any) in the input stream, with category codes 12 and 10 (for spaces). The functions differ in the case where the first character is a space: \str_tail:N and \str_tail:n only trim that space, while \str_tail_ignore_spaces:n removes the first non-space character and any space before it. If the ⟨*token list*⟩ is empty (or blank in the case of the _ignore_spaces variant), then nothing is left on the input stream.

\str_item:Nn ⋆
\str_item:cn ⋆
\str_item:nn ⋆
\str_item_ignore_spaces:nn ⋆

New: 2015-09-18

\str_item:nn {⟨*token list*⟩} {⟨*integer expression*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and leaves in the input stream the character in position ⟨*integer expression*⟩ of the ⟨*string*⟩, starting at 1 for the first (left-most) character. In the case of \str_item:Nn and \str_item:nn, all characters including spaces are taken into account. The \str_item_ignore_spaces:nn function skips spaces when counting characters. If the ⟨*integer expression*⟩ is negative, characters are counted from the end of the ⟨*string*⟩. Hence, -1 is the right-most character, *etc.*

```
\str_range:Nnn                 ⋆
\str_range:cnn                 ⋆
\str_range:nnn                 ⋆
\str_range_ignore_spaces:nnn   ⋆
```

New: 2015-09-18

`\str_range:nnn {⟨token list⟩} {⟨start index⟩} {⟨end index⟩}`

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and leaves in the input stream the characters from the ⟨*start index*⟩ to the ⟨*end index*⟩ inclusive. Spaces are preserved and counted as items (contrast this with `\tl_range:nnn` where spaces are not counted as items and are possibly discarded from the output).

Here ⟨*start index*⟩ and ⟨*end index*⟩ should be integer denotations. For describing in detail the functions' behavior, let m and n be the start and end index respectively. If either is 0, the result is empty. A positive index means 'start counting from the left end', a negative index means 'start counting from the right end'. Let l be the count of the token list.

The *actual start point* is determined as $M = m$ if $m > 0$ and as $M = l + m + 1$ if $m < 0$. Similarly the *actual end point* is $N = n$ if $n > 0$ and $N = l + n + 1$ if $n < 0$. If $M > N$, the result is empty. Otherwise it consists of all items from position M to position N inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions s for $s \leq 0$ or $s > l$. For instance,

```
\iow_term:e { \str_range:nnn { abcdef } { 2 } { 5 } }
\iow_term:e { \str_range:nnn { abcdef } { -4 } { -1 } }
\iow_term:e { \str_range:nnn { abcdef } { -2 } { -1 } }
\iow_term:e { \str_range:nnn { abcdef } { 0 } { -1 } }
```

prints `bcde`, `cdef`, `ef`, and an empty line to the terminal. The ⟨*start index*⟩ must always be smaller than or equal to the ⟨*end index*⟩: if this is not the case then no output is generated. Thus

```
\iow_term:e { \str_range:nnn { abcdef } { 5 } { 2 } }
\iow_term:e { \str_range:nnn { abcdef } { -1 } { -4 } }
```

both yield empty strings.

The behavior of `\str_range_ignore_spaces:nnn` is similar, but spaces are removed before starting the job. The input

```
\iow_term:e { \str_range:nnn { abcdefg } { 2 } { 5 } }
\iow_term:e { \str_range:nnn { abcdefg } { 2 } { -3 } }
\iow_term:e { \str_range:nnn { abcdefg } { -6 } { 5 } }
\iow_term:e { \str_range:nnn { abcdefg } { -6 } { -3 } }

\iow_term:e { \str_range:nnn { abc~efg } { 2 } { 5 } }
\iow_term:e { \str_range:nnn { abc~efg } { 2 } { -3 } }
\iow_term:e { \str_range:nnn { abc~efg } { -6 } { 5 } }
\iow_term:e { \str_range:nnn { abc~efg } { -6 } { -3 } }

\iow_term:e { \str_range_ignore_spaces:nnn { abcdefg } { 2 } { 5 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcdefg } { 2 } { -3 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcdefg } { -6 } { 5 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcdefg } { -6 } { -3 } }
```

```
\iow_term:e { \str_range_ignore_spaces:nnn { abcd~efg } { 2 } { 5 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcd~efg } { 2 } { -3 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcd~efg } { -6 } { 5 } }
\iow_term:e { \str_range_ignore_spaces:nnn { abcd~efg } { -6 } { -3 } }
```

will print four instances of `bcde`, four instances of `bc e` and eight instances of `bcde`.

17.6 Modifying string variables

`\str_replace_once:Nnn`
`\str_replace_once:cnn`
`\str_greplace_once:Nnn`
`\str_greplace_once:cnn`

New: 2017-10-08

`\str_replace_once:Nnn` ⟨*str var*⟩ {⟨*old*⟩} {⟨*new*⟩}

Converts the ⟨*old*⟩ and ⟨*new*⟩ token lists to strings, then replaces the first (leftmost) occurrence of ⟨*old string*⟩ in the ⟨*str var*⟩ with ⟨*new string*⟩.

`\str_replace_all:Nnn`
`\str_replace_all:cnn`
`\str_greplace_all:Nnn`
`\str_greplace_all:cnn`

New: 2017-10-08

`\str_replace_all:Nnn` ⟨*str var*⟩ {⟨*old*⟩} {⟨*new*⟩}

Converts the ⟨*old*⟩ and ⟨*new*⟩ token lists to strings, then replaces all occurrences of ⟨*old string*⟩ in the ⟨*str var*⟩ with ⟨*new string*⟩. As this function operates from left to right, the pattern ⟨*old string*⟩ may remain after the replacement (see `\str_remove_all:Nn` for an example).

`\str_remove_once:Nn`
`\str_remove_once:cn`
`\str_gremove_once:Nn`
`\str_gremove_once:cn`

New: 2017-10-08

`\str_remove_once:Nn` ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ then removes the first (leftmost) occurrence of ⟨*string*⟩ from the ⟨*str var*⟩.

`\str_remove_all:Nn`
`\str_remove_all:cn`
`\str_gremove_all:Nn`
`\str_gremove_all:cn`

New: 2017-10-08

`\str_remove_all:Nn` ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ then removes all occurrences of ⟨*string*⟩ from the ⟨*str var*⟩. As this function operates from left to right, the pattern ⟨*string*⟩ may remain after the removal, for instance,

```
\str_set:Nn \l_tmpa_str {abbccd} \str_remove_all:Nn \l_tmpa_str
{bc}
```

results in `\l_tmpa_str` containing `abcd`.

17.7 String manipulation

\str_lowercase:n ⋆
\str_lowercase:f ⋆
\str_uppercase:n ⋆
\str_uppercase:f ⋆

New: 2019-11-26

\str_lowercase:n {⟨*tokens*⟩}
\str_uppercase:n {⟨*tokens*⟩}

Converts the input ⟨*tokens*⟩ to their string representation, as described for `\tl_to_str:n`, and then to the lower or upper case representation using a one-to-one mapping as described by the Unicode Consortium file `UnicodeData.txt`.

These functions are intended for case changing programmatic data in places where upper/lower case distinctions are meaningful. One example would be automatically generating a function name from user input where some case changing is needed. In this situation the input is programmatic, not textual, case does have meaning and a language-independent one-to-one mapping is appropriate. For example

```
\cs_new_protected:Npn \myfunc:nn #1#2
  {
    \cs_set_protected:cpn
      {
        user
        \str_uppercase:f { \tl_head:n {#1} }
        \str_lowercase:f { \tl_tail:n {#1} }
      }
      { #2 }
  }
```

would be used to generate a function with an auto-generated name consisting of the upper case equivalent of the supplied name followed by the lower case equivalent of the rest of the input.

These functions should *not* be used for

- Caseless comparisons: use `\str_casefold:n` for this situation (case folding is distinct from lower casing).
- Case changing text for typesetting: see the `\text_lowercase:n(n)`, `\text_uppercase:n(n)` and `\text_titlecase_(all|first):n(n)` functions which correctly deal with context-dependence and other factors appropriate to text case changing.

\str_casefold:n ⋆
\str_casefold:V ⋆

New: 2022-10-16

\str_casefold:n {⟨*tokens*⟩}

Converts the input ⟨*tokens*⟩ to their string representation, as described for \tl_to_str:n, and then folds the case of the resulting ⟨*string*⟩ to remove case information. The result of this process is left in the input stream.

String folding is a process used for material such as identifiers rather than for "text". The folding provided by \str_casefold:n follows the mappings provided by the Unicode Consortium, who state:

> Case folding is primarily used for caseless comparison of text, such as identifiers in a computer program, rather than actual text transformation. Case folding in Unicode is based on the lowercase mapping, but includes additional changes to the source text to help make it language-insensitive and consistent. As a result, case-folded text should be used solely for internal processing and generally should not be stored or displayed to the end user.

The folding approach implemented by \str_casefold:n follows the "full" scheme defined by the Unicode Consortium (*e.g.* SS folds to SS). As case-folding is a language-insensitive process, there is no special treatment of Turkic input (*i.e.* I always folds to i and not to ı).

\str_mdfive_hash:n ⋆
\str_mdfive_hash:e ⋆

New: 2023-05-19

\str_mdfive_hash:n {⟨*tokens*⟩}

Expands to the MD5 sum generated from the ⟨*tokens*⟩, which is converted to a ⟨*string*⟩ as described for \tl_to_str:n.

17.8 Viewing strings

\str_show:N
\str_show:c
\str_show:n

New: 2015-09-18
Updated: 2021-04-29

\str_show:N ⟨*str var*⟩

Displays the content of the ⟨*str var*⟩ on the terminal.

\str_log:N
\str_log:c
\str_log:n

New: 2019-02-15
Updated: 2021-04-29

\str_log:N ⟨*str var*⟩

Writes the content of the ⟨*str var*⟩ in the log file.

17.9 Constant strings

`\c_ampersand_str`
`\c_atsign_str`
`\c_backslash_str`
`\c_left_brace_str`
`\c_right_brace_str`
`\c_circumflex_str`
`\c_colon_str`
`\c_dollar_str`
`\c_hash_str`
`\c_percent_str`
`\c_tilde_str`
`\c_underscore_str`
`\c_zero_str`

New: 2015-09-19
Updated: 2020-12-22

Constant strings, containing a single character token, with category code 12.

`\c_empty_str`

New: 2023-12-07

Constant that is always empty.

17.10 Scratch strings

`\l_tmpa_str`
`\l_tmpb_str`

Scratch strings for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_str`
`\g_tmpb_str`

Scratch strings for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Chapter 18

The l3str-convert module String encoding conversions

18.1 Encoding and escaping schemes

Traditionally, string encodings only specify how strings of characters should be stored as bytes. However, the resulting lists of bytes are often to be used in contexts where only a restricted subset of bytes are permitted (*e.g.*, PDF string objects, URLs). Hence, storing a string of characters is done in two steps.

- The code points ("character codes") are expressed as bytes following a given "encoding". This can be UTF-16, ISO 8859-1, *etc.* See Table 1 for a list of supported encodings.[6]
- Bytes are translated to TeX tokens through a given "escaping". Those are defined for the most part by the `pdf` file format. See Table 2 for a list of escaping methods supported.[6]

[6]Encodings and escapings will be added as they are requested.

Table 1: Supported encodings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the encoding in this list.

⟨*Encoding*⟩	description
`utf8`	UTF-8
`utf16`	UTF-16, with byte-order mark
`utf16be`	UTF-16, big-endian
`utf16le`	UTF-16, little-endian
`utf32`	UTF-32, with byte-order mark
`utf32be`	UTF-32, big-endian
`utf32le`	UTF-32, little-endian
`iso88591`, `latin1`	ISO 8859-1
`iso88592`, `latin2`	ISO 8859-2
`iso88593`, `latin3`	ISO 8859-3
`iso88594`, `latin4`	ISO 8859-4
`iso88595`	ISO 8859-5
`iso88596`	ISO 8859-6
`iso88597`	ISO 8859-7
`iso88598`	ISO 8859-8
`iso88599`, `latin5`	ISO 8859-9
`iso885910`, `latin6`	ISO 8859-10
`iso885911`	ISO 8859-11
`iso885913`, `latin7`	ISO 8859-13
`iso885914`, `latin8`	ISO 8859-14
`iso885915`, `latin9`	ISO 8859-15
`iso885916`, `latin10`	ISO 8859-16
`clist`	comma-list of integers
⟨*empty*⟩	native (Unicode) string
`default`	like `utf8` with 8-bit engines, and like native with unicode-engines

Table 2: Supported escapings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the escaping in this list.

⟨*Escaping*⟩	description
`bytes`, or empty	arbitrary bytes
`hex`, `hexadecimal`	byte = two hexadecimal digits
`name`	see `\pdfescapename`
`string`	see `\pdfescapestring`
`url`	encoding used in URLs

18.2 Conversion functions

`\str_set_convert:Nnnn`
`\str_gset_convert:Nnnn`

`\str_set_convert:Nnnn` ⟨*str var*⟩ {⟨*string*⟩} {⟨$name_1$⟩} {⟨$name_2$⟩}

This function converts the ⟨*string*⟩ from the encoding given by ⟨$name_1$⟩ to the encoding given by ⟨$name_2$⟩, and stores the result in the ⟨*str var*⟩. Each ⟨*name*⟩ can have the form ⟨*encoding*⟩ or ⟨*encoding*⟩/⟨*escaping*⟩, where the possible values of ⟨*encoding*⟩ and ⟨*escaping*⟩ are given in Tables 1 and 2, respectively. The default escaping is to input and output bytes directly. The special case of an empty ⟨*name*⟩ indicates the use of "native" strings, 8-bit for pdfTEX, and Unicode strings for the other two engines.

For example,

```
\str_set_convert:Nnnn \l_foo_str { Hello! } { } { utf16/hex }
```

results in the variable `\l_foo_str` holding the string `FEFF00480065006C006C006F0021`. This is obtained by converting each character in the (native) string `Hello!` to the UTF-16 encoding, and expressing each byte as a pair of hexadecimal digits. Note the presence of a (big-endian) byte order mark "FEFF, which can be avoided by specifying the encoding `utf16be/hex`.

An error is raised if the ⟨*string*⟩ is not valid according to the ⟨*escaping 1*⟩ and ⟨*encoding 1*⟩, or if it cannot be reencoded in the ⟨*encoding 2*⟩ and ⟨*escaping 2*⟩ (for instance, if a character does not exist in the ⟨*encoding 2*⟩). Erroneous input is replaced by the Unicode replacement character "FFFD, and characters which cannot be reencoded are replaced by either the replacement character "FFFD if it exists in the ⟨*encoding 2*⟩, or an encoding-specific replacement character, or the question mark character.

`\str_set_convert:Nnnn`*TF*
`\str_gset_convert:Nnnn`*TF*

`\str_set_convert:NnnnTF` ⟨*str var*⟩ {⟨*string*⟩} {⟨$name_1$⟩} {⟨$name_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

As `\str_set_convert:Nnnn`, converts the ⟨*string*⟩ from the encoding given by ⟨$name_1$⟩ to the encoding given by ⟨$name_2$⟩, and assigns the result to ⟨*str var*⟩. Contrarily to `\str_set_convert:Nnnn`, the conditional variant does not raise errors in case the ⟨*string*⟩ is not valid according to the ⟨$name_1$⟩ encoding, or cannot be expressed in the ⟨$name_2$⟩ encoding. Instead, the ⟨*false code*⟩ is performed.

18.3 Conversion by expansion (for PDF contexts)

A small number of expandable functions are provided for use in PDF string/name contexts. These *assume UTF-8* and *no escaping* in the input.

`\str_convert_pdfname:n` ⋆

`\str_convert_pdfname:n` {⟨*string*⟩}

As `\str_set_convert:Nnnn`, converts the ⟨*string*⟩ on a byte-by-byte basis with non-ASCII codepoints escaped using hashes.

18.4 Possibilities, and things to do

Encoding/escaping-related tasks.

- In XƎTEX/LuaTEX, would it be better to use the `^^^^....` approach to build a string from a given list of character codes? Namely, within a group, assign `0-9a-f` and all characters we want to category "other", then assign `^` the category superscript, and use `\scantokens`.
- Change `\str_set_convert:Nnnn` to expand its last two arguments.
- Describe the internal format in the code comments. Refuse code points in ["D800, "DFFF] in the internal representation?
- Add documentation about each encoding and escaping method, and add examples.
- The `hex` unescaping should raise an error for odd-token count strings.
- Decide what bytes should be escaped in the `url` escaping. Perhaps the characters `!'()*-./0123456789_` are safe, and all other characters should be escaped?
- Automate generation of 8-bit mapping files.
- Change the framework for 8-bit encodings: for decoding from 8-bit to Unicode, use 256 integer registers; for encoding, use a tree-box.
- More encodings (see Heiko's stringenc). CESU?
- More escapings: ASCII85, shell escapes, lua escapes, *etc.*?

Chapter 19

The l3quark module Quarks and scan marks

Two special types of constants in LaTeX3 are “quarks” and “scan marks”. By convention all constants of type quark start out with `\q_`, and scan marks start with `\s_`.

19.1 Quarks

Quarks are control sequences (and in fact, token lists) that expand to themselves and should therefore *never* be executed directly in the code. This would result in an endless loop!

They are meant to be used as delimiter in weird functions, the most common use case being the ‘stop token’ (*i.e.* `\q_stop`). For example, when writing a macro to parse a user-defined date

```
\date_parse:n {19/June/1981}
```

one might write a command such as

```
\cs_new:Npn \date_parse:n #1 { \date_parse_aux:w #1 \q_stop }
\cs_new:Npn \date_parse_aux:w #1 / #2 / #3 \q_stop
  { <do something with the date> }
```

Quarks are sometimes also used as error return values for functions that receive erroneous input. For example, in the function `\prop_get:NnN` to retrieve a value stored in some key of a property list, if the key does not exist then the return value is the quark `\q_no_value`. As mentioned above, such quarks are extremely fragile and it is imperative when using such functions that code is carefully written to check for pathological cases to avoid leakage of a quark into an uncontrolled environment.

Quarks also permit the following ingenious trick when parsing tokens: when you pick up a token in a temporary variable and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary variable to the quark using `\tl_if_eq:NNTF`. A set of special quark testing functions is set up below. All the quark testing functions are expandable although the ones testing only single tokens are much faster.

19.2 Defining quarks

`\quark_new:N` `\quark_new:N` ⟨*quark*⟩

Creates a new ⟨*quark*⟩ which expands only to ⟨*quark*⟩. The ⟨*quark*⟩ is defined globally, and an error message is raised if the name was already taken.

`\q_stop` Used as a marker for delimited arguments, such as

```
\cs_set:Npn \tmp:w #1#2 \q_stop {#1}
```

`\q_mark` Used as a marker for delimited arguments when `\q_stop` is already in use.

`\q_nil` Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself need to be tested (in contrast to `\q_stop`, which is only ever used as a delimiter).

`\q_no_value` A canonical value for a missing value, when one is requested from a data structure. This is therefore used as a "return" value by functions such as `\prop_get:NnN` if there is no data to return.

19.3 Quark tests

The method used to define quarks means that the single token (`N`) tests are faster than the multi-token (`n`) tests. The latter should therefore only be used when the argument can definitely take more than a single token.

`\quark_if_nil_p:N` ⋆
`\quark_if_nil:NTF` ⋆

`\quark_if_nil_p:N` ⟨*token*⟩
`\quark_if_nil:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is equal to `\q_nil`.

`\quark_if_nil_p:n` ⋆
`\quark_if_nil_p:(o|V)` ⋆
`\quark_if_nil:nTF` ⋆
`\quark_if_nil:(o|V)TF` ⋆

`\quark_if_nil_p:n` {⟨*token list*⟩}
`\quark_if_nil:nTF` {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list*⟩ contains only `\q_nil` (distinct from ⟨*token list*⟩ being empty or containing `\q_nil` plus one or more other tokens).

`\quark_if_no_value_p:N` ⋆
`\quark_if_no_value_p:c` ⋆
`\quark_if_no_value:NTF` ⋆
`\quark_if_no_value:cTF` ⋆

`\quark_if_no_value_p:N` ⟨*token*⟩
`\quark_if_no_value:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is equal to `\q_no_value`.

`\quark_if_no_value_p:n` ⋆
`\quark_if_no_value:nTF` ⋆

`\quark_if_no_value_p:n` {⟨*token list*⟩}
`\quark_if_no_value:nTF` {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list*⟩ contains only `\q_no_value` (distinct from ⟨*token list*⟩ being empty or containing `\q_no_value` plus one or more other tokens).

19.4 Recursion

This module provides a uniform interface to intercepting and terminating loops as when one is doing tail recursion. The building blocks follow below and an example is shown in Section 19.4.1.

`\q_recursion_tail`

This quark is appended to the data structure in question and appears as a real element there. This means it gets any list separators around it.

`\q_recursion_stop`

This quark is added *after* the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.

`\quark_if_recursion_tail_stop:N` ⋆

`\quark_if_recursion_tail_stop:N` ⟨*token*⟩

Tests if ⟨*token*⟩ contains only the marker `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

`\quark_if_recursion_tail_stop:n` ⋆
`\quark_if_recursion_tail_stop:o` ⋆

Updated: 2011-09-06

`\quark_if_recursion_tail_stop:n` {⟨*token list*⟩}

Tests if the ⟨*token list*⟩ contains only `\q_recursion_tail`, and if so uses `\use_none_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items.

`\quark_if_recursion_tail_stop_do:Nn` ⋆

`\quark_if_recursion_tail_stop_do:Nn` ⟨*token*⟩ {⟨*insertion*⟩}

Tests if ⟨*token*⟩ contains only the marker `\q_recursion_tail`, and if so uses `\use_i_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The ⟨*insertion*⟩ code is then added to the input stream after the recursion has ended.

`\quark_if_recursion_tail_stop_do:nn` ⋆
`\quark_if_recursion_tail_stop_do:on` ⋆

Updated: 2011-09-06

`\quark_if_recursion_tail_stop_do:nn` {⟨*token list*⟩} {⟨*insertion*⟩}

Tests if the ⟨*token list*⟩ contains only `\q_recursion_tail`, and if so uses `\use_i_delimit_by_q_recursion_stop:w` to terminate the recursion that this belongs to. The recursion input must include the marker tokens `\q_recursion_tail` and `\q_recursion_stop` as the last two items. The ⟨*insertion*⟩ code is then added to the input stream after the recursion has ended.

```
\quark_if_recursion_tail_break:NN ⋆
\quark_if_recursion_tail_break:nN ⋆
```

New: 2018-04-10

`\quark_if_recursion_tail_break:nN {⟨token list⟩} \⟨type⟩_map_break:`

Tests if ⟨*token list*⟩ contains only `\q_recursion_tail`, and if so terminates the recursion using `\⟨type⟩_map_break:`. The recursion end should be marked by `\prg_break_point:Nn \⟨type⟩_map_break:`.

19.4.1 An example of recursion with quarks

Quarks are mainly used internally in the expl3 code to define recursion functions such as `\tl_map_inline:nn` and so on. Here is a small example to demonstrate how to use quarks in this fashion. We shall define a command called `\my_map_dbl:nn` which takes a token list and applies an operation to every *pair* of tokens. For example, `\my_map_dbl:nn {abcd} {[--#1--#2--]~}` would produce "`[-a-b-] [-c-d-] `". Using quarks to define such functions simplifies their logic and ensures robustness in many cases.

Here's the definition of `\my_map_dbl:nn`. First of all, define the function that does the processing based on the inline function argument `#2`. Then initiate the recursion using an internal function. The token list `#1` is terminated using `\q_recursion_tail`, with delimiters according to the type of recursion (here a pair of `\q_recursion_tail`), concluding with `\q_recursion_stop`. These quarks are used to mark the end of the token list being operated upon.

```
\cs_new:Npn \my_map_dbl:nn #1#2
 {
   \cs_set:Npn \__my_map_dbl_fn:nn ##1 ##2 {#2}
   \__my_map_dbl:nn #1 \q_recursion_tail \q_recursion_tail
   \q_recursion_stop
 }
```

The definition of the internal recursion function follows. First check if either of the input tokens are the termination quarks. Then, if not, apply the inline function to the two arguments.

```
\cs_new:Nn \__my_map_dbl:nn
 {
   \quark_if_recursion_tail_stop:n {#1}
   \quark_if_recursion_tail_stop:n {#2}
   \__my_map_dbl_fn:nn {#1} {#2}
```

Finally, recurse:

```
   \__my_map_dbl:nn
 }
```

Note that contrarily to LaTeX3 built-in mapping functions, this mapping function cannot be nested, since the second map would overwrite the definition of `\__my_map_dbl_fn:nn`.

19.5 Scan marks

Scan marks are control sequences set equal to `\scan_stop:`, hence never expand in an expansion context and are (largely) invisible if they are encountered in a typesetting context.

Like quarks, they can be used as delimiters in weird functions and are often safer to use for this purpose. Since they are harmless when executed by TeX in non-expandable contexts, they can be used to mark the end of a set of instructions. This allows to skip to that point if the end of the instructions should not be performed (see l3regex).

`\scan_new:N` `\scan_new:N` ⟨*scan mark*⟩

New: 2018-04-01

Creates a new ⟨*scan mark*⟩ which is set equal to `\scan_stop:`. The ⟨*scan mark*⟩ is defined globally, and an error message is raised if the name was already taken by another scan mark.

`\s_stop`

New: 2018-04-01

Used at the end of a set of instructions, as a marker that can be jumped to using `\use_none_delimit_by_s_stop:w`.

`\use_none_delimit_by_s_stop:w` ⋆ `\use_none_delimit_by_s_stop:w` ⟨*tokens*⟩ `\s_stop`

New: 2018-04-01

Removes the ⟨*tokens*⟩ and `\s_stop` from the input stream. This leads to a low-level TeX error if `\s_stop` is absent.

Chapter 20

The l3seq module Sequences and stacks

LaTeX3 implements a "sequence" data type, which contain an ordered list of entries which may contain any ⟨*balanced text*⟩. It is possible to map functions to sequences such that the function is applied to every item in the sequence.

Sequences are also used to implement stack functions in LaTeX3. This is achieved using a number of dedicated stack functions.

20.1 Creating and initialising sequences

`\seq_new:N`
`\seq_new:c`

`\seq_new:N` ⟨*seq var*⟩

Creates a new ⟨*seq var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*seq var*⟩ initially contains no items.

`\seq_clear:N`
`\seq_clear:c`
`\seq_gclear:N`
`\seq_gclear:c`

`\seq_clear:N` ⟨*seq var*⟩

Clears all items from the ⟨*seq var*⟩.

`\seq_clear_new:N`
`\seq_clear_new:c`
`\seq_gclear_new:N`
`\seq_gclear_new:c`

`\seq_clear_new:N` ⟨*seq var*⟩

Ensures that the ⟨*seq var*⟩ exists globally by applying `\seq_new:N` if necessary, then applies `\seq_(g)clear:N` to leave the ⟨*seq var*⟩ empty.

`\seq_set_eq:NN`
`\seq_set_eq:(cN|Nc|cc)`
`\seq_gset_eq:NN`
`\seq_gset_eq:(cN|Nc|cc)`

`\seq_set_eq:NN` ⟨*seq var*$_1$⟩ ⟨*seq var*$_2$⟩

Sets the content of ⟨*seq var*$_1$⟩ equal to that of ⟨*seq var*$_2$⟩.

`\seq_set_from_clist:NN`
`\seq_set_from_clist:(cN|Nc|cc)`
`\seq_set_from_clist:Nn`
`\seq_set_from_clist:cn`
`\seq_gset_from_clist:NN`
`\seq_gset_from_clist:(cN|Nc|cc)`
`\seq_gset_from_clist:Nn`
`\seq_gset_from_clist:cn`

New: 2014-07-17

`\seq_set_from_clist:NN` ⟨*seq var*⟩ ⟨*comma-list*⟩

Converts the data in the ⟨*comma list*⟩ into a ⟨*seq var*⟩: the original ⟨*comma list*⟩ is unchanged.

`\seq_const_from_clist:Nn`
`\seq_const_from_clist:cn`

New: 2017-11-28

`\seq_const_from_clist:Nn` ⟨*seq var*⟩ {⟨*comma-list*⟩}

Creates a new constant ⟨*seq var*⟩ or raises an error if the name is already taken. The ⟨*seq var*⟩ is set globally to contain the items in the ⟨*comma list*⟩.

`\seq_set_split:Nnn`
`\seq_set_split:(NVn|NnV|NVV|Nne|Nee)`
`\seq_gset_split:Nnn`
`\seq_gset_split:(NVn|NnV|NVV|Nne|Nee)`

New: 2011-08-15
Updated: 2012-07-02

`\seq_set_split:Nnn` ⟨*seq var*⟩ {⟨*delimiter*⟩} {⟨*token list*⟩}

Splits the ⟨*token list*⟩ into ⟨*items*⟩ separated by ⟨*delimiter*⟩, and assigns the result to the ⟨*seq var*⟩. Spaces on both sides of each ⟨*item*⟩ are ignored, then one set of outer braces is removed (if any); this space trimming behaviour is identical to that of l3clist functions. Empty ⟨*items*⟩ are preserved by `\seq_set_split:Nnn`, and can be removed afterwards using `\seq_remove_all:Nn` ⟨*seq var*⟩ `{}`. The ⟨*delimiter*⟩ may not contain `{`, `}` or `#` (assuming TeX's normal category code régime). If the ⟨*delimiter*⟩ is empty, the ⟨*token list*⟩ is split into ⟨*items*⟩ as a ⟨*token list*⟩. See also `\seq_set_split_keep_spaces:Nnn`, which omits space stripping.

`\seq_set_split_keep_spaces:Nnn`
`\seq_set_split_keep_spaces:NnV`
`\seq_gset_split_keep_spaces:Nnn`
`\seq_gset_split_keep_spaces:NnV`

New: 2021-03-24

`\seq_set_split_keep_spaces:Nnn` ⟨*seq var*⟩ {⟨*delimiter*⟩} {⟨*token list*⟩}

Splits the ⟨*token list*⟩ into ⟨*items*⟩ separated by ⟨*delimiter*⟩, and assigns the result to the ⟨*seq var*⟩. One set of outer braces is removed (if any) but any surrounding spaces are retained: any braces *inside* one or more spaces are therefore kept. Empty ⟨*items*⟩ are preserved by `\seq_set_split_keep_spaces:Nnn`, and can be removed afterwards using `\seq_remove_all:Nn` ⟨*seq var*⟩ `{}`. The ⟨*delimiter*⟩ may not contain `{`, `}` or `#` (assuming TeX's normal category code régime). If the ⟨*delimiter*⟩ is empty, the ⟨*token list*⟩ is split into ⟨*items*⟩ as a ⟨*token list*⟩. See also `\seq_set_split:Nnn`, which removes spaces around the delimiters.

\seq_set_filter:NNn
\seq_gset_filter:NNn

New: 2012-06-15

\seq_set_filter:NNn ⟨*seq var*$_1$⟩ ⟨*seq var*$_2$⟩ {⟨*inline boolexpr*⟩}

Evaluates the ⟨*inline boolexpr*⟩ for every ⟨*item*⟩ stored within the ⟨*seq var*$_2$⟩. The ⟨*inline boolexpr*⟩ receives the ⟨*item*⟩ as #1. The sequence of all ⟨*items*⟩ for which the ⟨*inline boolexpr*⟩ evaluated to true is assigned to ⟨*seq var*$_1$⟩.

TEXhackers note: Contrarily to other mapping functions, \seq_map_break: cannot be used in this function, and would lead to low-level TEX errors.

\seq_concat:NNN
\seq_concat:ccc
\seq_gconcat:NNN
\seq_gconcat:ccc

\seq_concat:NNN ⟨*seq var*$_1$⟩ ⟨*seq var*$_2$⟩ ⟨*seq var*$_3$⟩

Concatenates the content of ⟨*seq var*$_2$⟩ and ⟨*seq var*$_3$⟩ together and saves the result in ⟨*seq var*$_1$⟩. The items in ⟨*seq var*$_2$⟩ are placed at the left side of the new sequence.

\seq_if_exist_p:N ⋆
\seq_if_exist_p:c ⋆
\seq_if_exist:N*TF* ⋆
\seq_if_exist:c*TF* ⋆

New: 2012-03-03

\seq_if_exist_p:N ⟨*seq var*⟩
\seq_if_exist:NTF ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*seq var*⟩ is currently defined. This does not check that the ⟨*seq var*⟩ really is a sequence variable.

20.2 Appending data to sequences

\seq_put_left:Nn
\seq_put_left:(NV|Nv|Ne|No|cn|cV|cv|ce|co)
\seq_gput_left:Nn
\seq_gput_left:(NV|Nv|Ne|No|cn|cV|cv|ce|co)

\seq_put_left:Nn ⟨*seq var*⟩ {⟨*item*⟩}

Appends the ⟨*item*⟩ to the left of the ⟨*seq var*⟩.

\seq_put_right:Nn
\seq_put_right:(NV|Nv|Ne|No|cn|cV|cv|ce|co)
\seq_gput_right:Nn
\seq_gput_right:(NV|Nv|Ne|No|cn|cV|cv|ce|co)

\seq_put_right:Nn ⟨*seq var*⟩ {⟨*item*⟩}

Appends the ⟨*item*⟩ to the right of the ⟨*seq var*⟩.

20.3 Recovering items from sequences

Items can be recovered from either the left or the right of sequences. For implementation reasons, the actions at the left of the sequence are faster than those acting on the right. These functions all assign the recovered material locally, *i.e.* setting the ⟨*tl var*⟩ used with \tl_set:Nn and *never* \tl_gset:Nn.

\seq_get_left:NN
\seq_get_left:cN

Updated: 2012-05-14

\seq_get_left:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Stores the left-most item from a ⟨*seq var*⟩ in the ⟨*tl var*⟩ without removing it from the ⟨*seq var*⟩. The ⟨*tl var*⟩ is assigned locally. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker \q_no_value.

\seq_get_right:NN
\seq_get_right:cN

Updated: 2012-05-19

\seq_get_right:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Stores the right-most item from a ⟨*seq var*⟩ in the ⟨*tl var*⟩ without removing it from the ⟨*seq var*⟩. The ⟨*tl var*⟩ is assigned locally. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker \q_no_value.

\seq_pop_left:NN
\seq_pop_left:cN

Updated: 2012-05-14

\seq_pop_left:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Pops the left-most item from a ⟨*seq var*⟩ into the ⟨*tl var*⟩, *i.e.* removes the item from the sequence and stores it in the ⟨*tl var*⟩. Both of the variables are assigned locally. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker \q_no_value.

\seq_gpop_left:NN
\seq_gpop_left:cN

Updated: 2012-05-14

\seq_gpop_left:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Pops the left-most item from a ⟨*seq var*⟩ into the ⟨*tl var*⟩, *i.e.* removes the item from the sequence and stores it in the ⟨*tl var*⟩. The ⟨*seq var*⟩ is modified globally, while the assignment of the ⟨*tl var*⟩ is local. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker \q_no_value.

\seq_pop_right:NN
\seq_pop_right:cN

Updated: 2012-05-19

\seq_pop_right:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Pops the right-most item from a ⟨*seq var*⟩ into the ⟨*tl var*⟩, *i.e.* removes the item from the sequence and stores it in the ⟨*tl var*⟩. Both of the variables are assigned locally. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker \q_no_value.

\seq_gpop_right:NN
\seq_gpop_right:cN

Updated: 2012-05-19

\seq_gpop_right:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Pops the right-most item from a ⟨*seq var*⟩ into the ⟨*tl var*⟩, *i.e.* removes the item from the sequence and stores it in the ⟨*tl var*⟩. The ⟨*seq var*⟩ is modified globally, while the assignment of the ⟨*tl var*⟩ is local. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker \q_no_value.

\seq_item:Nn ⋆
\seq_item:(NV|Ne|cn|cV|ce) ⋆

New: 2014-07-17

\seq_item:Nn ⟨*seq var*⟩ {⟨*integer expression*⟩}

Indexing items in the ⟨*seq var*⟩ from 1 at the top (left), this function evaluates the ⟨*integer expression*⟩ and leaves the appropriate item from the sequence in the input stream. If the ⟨*integer expression*⟩ is negative, indexing occurs from the bottom (right) of the sequence. If the ⟨*integer expression*⟩ is larger than the number of items in the ⟨*seq var*⟩ (as calculated by \seq_count:N) then the function expands to nothing.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨*item*⟩ does not expand further when appearing in an e-type or x-type argument expansion.

\seq_rand_item:N ⋆
\seq_rand_item:c ⋆

New: 2016-12-06

\seq_rand_item:N ⟨*seq var*⟩

Selects a pseudo-random item of the ⟨*seq var*⟩. If the ⟨*seq var*⟩ is empty the result is empty.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨*item*⟩ does not expand further when appearing in an e-type or x-type argument expansion.

20.4 Recovering values from sequences with branching

The functions in this section combine tests for non-empty sequences with recovery of an item from the sequence. They offer increased readability and performance over separate testing and recovery phases.

\seq_get_left:NNTF
\seq_get_left:cNTF
New: 2012-05-14
Updated: 2012-05-19

`\seq_get_left:NNTF ⟨seq var⟩ ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}`

If the ⟨*seq var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*seq var*⟩ is non-empty, stores the left-most item from the ⟨*seq var*⟩ in the ⟨*tl var*⟩ without removing it from the ⟨*seq var*⟩, then leaves the ⟨*true code*⟩ in the input stream. The ⟨*tl var*⟩ is assigned locally.

\seq_get_right:NNTF
\seq_get_right:cNTF
New: 2012-05-19

`\seq_get_right:NNTF ⟨seq var⟩ ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}`

If the ⟨*seq var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*seq var*⟩ is non-empty, stores the right-most item from the ⟨*seq var*⟩ in the ⟨*tl var*⟩ without removing it from the ⟨*seq var*⟩, then leaves the ⟨*true code*⟩ in the input stream. The ⟨*tl var*⟩ is assigned locally.

\seq_pop_left:NNTF
\seq_pop_left:cNTF
New: 2012-05-14
Updated: 2012-05-19

`\seq_pop_left:NNTF ⟨seq var⟩ ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}`

If the ⟨*seq var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*seq var*⟩ is non-empty, pops the left-most item from the ⟨*seq var*⟩ in the ⟨*tl var*⟩, *i.e.* removes the item from the ⟨*seq var*⟩, then leaves the ⟨*true code*⟩ in the input stream. Both the ⟨*seq var*⟩ and the ⟨*tl var*⟩ are assigned locally.

\seq_gpop_left:NNTF
\seq_gpop_left:cNTF
New: 2012-05-14
Updated: 2012-05-19

`\seq_gpop_left:NNTF ⟨seq var⟩ ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}`

If the ⟨*seq var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*seq var*⟩ is non-empty, pops the left-most item from the ⟨*seq var*⟩ in the ⟨*tl var*⟩, *i.e.* removes the item from the ⟨*seq var*⟩, then leaves the ⟨*true code*⟩ in the input stream. The ⟨*seq var*⟩ is modified globally, while the ⟨*tl var*⟩ is assigned locally.

\seq_pop_right:NNTF
\seq_pop_right:cNTF
New: 2012-05-19

`\seq_pop_right:NNTF ⟨seq var⟩ ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}`

If the ⟨*seq var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*seq var*⟩ is non-empty, pops the right-most item from the ⟨*seq var*⟩ in the ⟨*tl var*⟩, *i.e.* removes the item from the ⟨*seq var*⟩, then leaves the ⟨*true code*⟩ in the input stream. Both the ⟨*seq var*⟩ and the ⟨*tl var*⟩ are assigned locally.

\seq_gpop_right:NNTF
\seq_gpop_right:cNTF
New: 2012-05-19

`\seq_gpop_right:NNTF ⟨seq var⟩ ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}`

If the ⟨*seq var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*seq var*⟩ is non-empty, pops the right-most item from the ⟨*seq var*⟩ in the ⟨*tl var*⟩, *i.e.* removes the item from the ⟨*seq var*⟩, then leaves the ⟨*true code*⟩ in the input stream. The ⟨*seq var*⟩ is modified globally, while the ⟨*tl var*⟩ is assigned locally.

20.5 Modifying sequences

While sequences are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update sequences, while retaining the order of the unaffected entries.

\seq_remove_duplicates:N
\seq_remove_duplicates:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c

\seq_remove_duplicates:N ⟨seq var⟩

Removes duplicate items from the ⟨seq var⟩, leaving the left most copy of each item in the ⟨seq var⟩. The ⟨item⟩ comparison takes place on a token basis, as for \tl_if_eq:nnTF.

TeXhackers note: This function iterates through every item in the ⟨seq var⟩ and does a comparison with the ⟨items⟩ already checked. It is therefore relatively slow with large sequences.

\seq_remove_all:Nn
\seq_remove_all:(NV|Ne|cn|cV|ce)
\seq_gremove_all:Nn
\seq_gremove_all:(NV|Ne|cn|cV|ce)

\seq_remove_all:Nn ⟨seq var⟩ {⟨item⟩}

Removes every occurrence of ⟨item⟩ from the ⟨seq var⟩. The ⟨item⟩ comparison takes place on a token basis, as for \tl_if_eq:nnTF.

\seq_set_item:Nnn
\seq_set_item:cnn
\seq_set_item:Nnn*TF*
\seq_set_item:cnn*TF*
\seq_gset_item:Nnn
\seq_gset_item:cnn
\seq_gset_item:Nnn*TF*
\seq_gset_item:cnn*TF*

New: 2021-04-29

\seq_set_item:Nnn ⟨seq var⟩ {⟨int expr⟩} {⟨item⟩}
\seq_set_item:NnnTF ⟨seq var⟩ {⟨int expr⟩} {⟨item⟩} {⟨true code⟩} {⟨false code⟩}

Removes the item of ⟨seq var⟩ at the position given by evaluating the ⟨int expr⟩ and replaces it by ⟨item⟩. Items are indexed from 1 on the left/top of the ⟨seq var⟩, or from -1 on the right/bottom. If the ⟨int expr⟩ is zero or is larger (in absolute value) than the number of items in the sequence, the ⟨seq var⟩ is not modified. In these cases, \seq_set_item:Nnn raises an error while \seq_set_item:NnnTF runs the ⟨false code⟩. In cases where the assignment was successful, ⟨true code⟩ is run afterwards.

\seq_reverse:N
\seq_reverse:c
\seq_greverse:N
\seq_greverse:c

New: 2014-07-18

\seq_reverse:N ⟨seq var⟩

Reverses the order of the items stored in the ⟨seq var⟩.

\seq_sort:Nn
\seq_sort:cn
\seq_gsort:Nn
\seq_gsort:cn

New: 2017-02-06

\seq_sort:Nn ⟨seq var⟩ {⟨comparison code⟩}

Sorts the items in the ⟨seq var⟩ according to the ⟨comparison code⟩, and assigns the result to ⟨seq var⟩. The details of sorting comparison are described in Section 6.1.

\seq_shuffle:N
\seq_shuffle:c
\seq_gshuffle:N
\seq_gshuffle:c

New: 2018-04-29

\seq_shuffle:N ⟨*seq var*⟩

Sets the ⟨*seq var*⟩ to the result of placing the items of the ⟨*seq var*⟩ in a random order. Each item is (roughly) as likely to end up in any given position.

TEXhackers note: For sequences with more than 13 items or so, only a small proportion of all possible permutations can be reached, because the random seed \sys_rand_seed: only has 28-bits. The use of \toks internally means that sequences with more than 32767 or 65535 items (depending on the engine) cannot be shuffled.

20.6 Sequence conditionals

\seq_if_empty_p:N ⋆
\seq_if_empty_p:c ⋆
\seq_if_empty:N*TF* ⋆
\seq_if_empty:c*TF* ⋆

\seq_if_empty_p:N ⟨*seq var*⟩
\seq_if_empty:NTF ⟨*seq var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*seq var*⟩ is empty (containing no items).

\seq_if_in:Nn*TF*
\seq_if_in:(NV|Nv|Ne|No|cn|cV|cv|ce|co)*TF*

\seq_if_in:NnTF ⟨*seq var*⟩ {⟨*item*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*item*⟩ is present in the ⟨*seq var*⟩.

20.7 Mapping over sequences

All mappings are done at the current group level, *i.e.* any local assignments made by the ⟨*function*⟩ or ⟨*code*⟩ discussed below remain in effect after the loop.

\seq_map_function:NN ☆
\seq_map_function:cN ☆

Updated: 2012-06-29

\seq_map_function:NN ⟨*seq var*⟩ ⟨*function*⟩

Applies ⟨*function*⟩ to every ⟨*item*⟩ stored in the ⟨*seq var*⟩. The ⟨*function*⟩ will receive one argument for each iteration. The ⟨*items*⟩ are returned from left to right. To pass further arguments to the ⟨*function*⟩, see \seq_map_tokens:Nn. The function \seq_map_inline:Nn is faster than \seq_map_function:NN for sequences with more than about 10 items.

\seq_map_inline:Nn
\seq_map_inline:cn

Updated: 2012-06-29

\seq_map_inline:Nn ⟨*seq var*⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*seq var*⟩. The ⟨*inline function*⟩ should consist of code which will receive the ⟨*item*⟩ as #1. The ⟨*items*⟩ are returned from left to right.

\seq_map_tokens:Nn ☆
\seq_map_tokens:cn ☆

New: 2019-08-30

\seq_map_tokens:Nn ⟨seq var⟩ {⟨code⟩}

Analogue of \seq_map_function:NN which maps several tokens instead of a single function. The ⟨code⟩ receives each item in the ⟨seq var⟩ as a trailing brace group. For instance,

```
\seq_map_tokens:Nn \l_my_seq { \prg_replicate:nn { 2 } }
```

expands to twice each item in the ⟨seq var⟩: for each item in \l_my_seq the function \prg_replicate:nn receives 2 and ⟨item⟩ as its two arguments. The function \seq_map_inline:Nn is typically faster but it is not expandable.

\seq_map_variable:NNn
\seq_map_variable:(Ncn|cNn|ccn)

Updated: 2012-06-29

\seq_map_variable:NNn ⟨seq var⟩ ⟨variable⟩ {⟨code⟩}

Stores each ⟨item⟩ of the ⟨seq var⟩ in turn in the (token list) ⟨variable⟩ and applies the ⟨code⟩. The ⟨code⟩ will usually make use of the ⟨variable⟩, but this is not enforced. The assignments to the ⟨variable⟩ are local. Its value after the loop is the last ⟨item⟩ in the ⟨seq var⟩, or its original value if the ⟨seq var⟩ is empty. The ⟨items⟩ are returned from left to right.

\seq_map_indexed_function:NN ☆

New: 2018-05-03

\seq_map_indexed_function:NN ⟨seq var⟩ ⟨function⟩

Applies ⟨function⟩ to every entry in the ⟨seq var⟩. The ⟨function⟩ should have signature :nn. It receives two arguments for each iteration: the ⟨index⟩ (namely 1 for the first entry, then 2 and so on) and the ⟨item⟩.

\seq_map_indexed_inline:Nn

New: 2018-05-03

\seq_map_indexed_inline:Nn ⟨seq var⟩ {⟨inline function⟩}

Applies ⟨inline function⟩ to every entry in the ⟨seq var⟩. The ⟨inline function⟩ should consist of code which receives the ⟨index⟩ (namely 1 for the first entry, then 2 and so on) as #1 and the ⟨item⟩ as #2.

\seq_map_pairwise_function:NNN ☆
\seq_map_pairwise_function:(NcN|cNN|ccN) ☆

New: 2023-05-10

\seq_map_pairwise_function:NNN ⟨seq_1⟩ ⟨seq_2⟩ ⟨function⟩

Applies ⟨function⟩ to every pair of items ⟨seq_1-item⟩–⟨seq_2-item⟩ from the two sequences, returning items from both sequences from left to right. The ⟨function⟩ receives two n-type arguments for each iteration. The mapping terminates when the end of either sequence is reached (*i.e.* whichever sequence has fewer items determines how many iterations occur).

`\seq_map_break:` ☆

Updated: 2012-06-29

`\seq_map_break:`

Used to terminate a `\seq_map_...` function before all entries in the ⟨*seq var*⟩ have been processed. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \seq_map_break: }
      {
        % Do something useful
      }
  }
```

Use outside of a `\seq_map_...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

`\seq_map_break:n` ☆

Updated: 2012-06-29

`\seq_map_break:n {`⟨*code*⟩`}`

Used to terminate a `\seq_map_...` function before all entries in the ⟨*seq var*⟩ have been processed, inserting the ⟨*code*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \seq_map_break:n { <code> } }
      {
        % Do something useful
      }
  }
```

Use outside of a `\seq_map_...` scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the ⟨*code*⟩ is inserted into the input stream. This depends on the design of the mapping function.

`\seq_set_map:NNn`
`\seq_gset_map:NNn`

New: 2011-12-22
Updated: 2020-07-16

`\seq_set_map:NNn` ⟨*seq var*$_1$⟩ ⟨*seq var*$_2$⟩ `{`⟨*inline function*⟩`}`

Applies ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*seq var*$_2$⟩. The ⟨*inline function*⟩ should consist of code which will receive the ⟨*item*⟩ as `#1`. The sequence resulting from applying ⟨*inline function*⟩ to each ⟨*item*⟩ is assigned to ⟨*seq var*$_1$⟩.

TeXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level TeX errors.

`\seq_set_map_e:NNn`
`\seq_gset_map_e:NNn`

New: 2020-07-16
Updated: 2023-10-26

`\seq_set_map_e:NNn` ⟨*seq var*$_1$⟩ ⟨*seq var*$_2$⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*seq var*$_2$⟩. The ⟨*inline function*⟩ should consist of code which will receive the ⟨*item*⟩ as `#1`. The sequence resulting from `e`-expanding ⟨*inline function*⟩ applied to each ⟨*item*⟩ is assigned to ⟨*seq var*$_1$⟩. As such, the code in ⟨*inline function*⟩ should be expandable.

TEXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level TEX errors.

`\seq_count:N` ⋆
`\seq_count:c` ⋆

New: 2012-07-13

`\seq_count:N` ⟨*seq var*⟩

Leaves the number of items in the ⟨*seq var*⟩ in the input stream as an ⟨*integer denotation*⟩. The total number of items in a ⟨*seq var*⟩ includes those which are empty and duplicates, *i.e.* every item in a ⟨*seq var*⟩ is unique.

20.8 Using the content of sequences directly

`\seq_use:Nnnn` ⋆
`\seq_use:cnnn` ⋆

New: 2013-05-26

`\seq_use:Nnnn` ⟨*seq var*⟩ {⟨*separator between two*⟩}
{⟨*separator between more than two*⟩} {⟨*separator between final two*⟩}

Places the contents of the ⟨*seq var*⟩ in the input stream, with the appropriate ⟨*separator*⟩ between the items. Namely, if the sequence has more than two items, the ⟨*separator between more than two*⟩ is placed between each pair of items except the last, for which the ⟨*separator between final two*⟩ is used. If the sequence has exactly two items, then they are placed in the input stream separated by the ⟨*separator between two*⟩. If the sequence has a single item, it is placed in the input stream, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nnnn \l_tmpa_seq { ~and~ } { ,~ } { ,~and~ }
```

inserts "`a, b, c, de, and f`" in the input stream. The first separator argument is not used in this case because the sequence has more than 2 items.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*items*⟩ do not expand further when appearing in an `e`-type or `x`-type argument expansion.

\seq_use:Nn ⋆
\seq_use:cn ⋆

New: 2013-05-26

\seq_use:Nn ⟨*seq var*⟩ {⟨*separator*⟩}

Places the contents of the ⟨*seq var*⟩ in the input stream, with the ⟨*separator*⟩ between the items. If the sequence has a single item, it is placed in the input stream with no ⟨*separator*⟩, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nn \l_tmpa_seq { ~and~ }
```

inserts "`a and b and c and de and f`" in the input stream.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*items*⟩ do not expand further when appearing in an `e`-type or `x`-type argument expansion.

20.9 Sequences as stacks

Sequences can be used as stacks, where data is pushed to and popped from the top of the sequence. (The left of a sequence is the top, for performance reasons.) The stack functions for sequences are not intended to be mixed with the general ordered data functions detailed in the previous section: a sequence should either be used as an ordered data type or as a stack, but not in both ways.

\seq_get:NN
\seq_get:cN

Updated: 2012-05-14

\seq_get:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Reads the top item from a ⟨*seq var*⟩ into the ⟨*tl var*⟩ without removing it from the ⟨*seq var*⟩. The ⟨*tl var*⟩ is assigned locally. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker `\q_no_value`.

\seq_pop:NN
\seq_pop:cN

Updated: 2012-05-14

\seq_pop:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Pops the top item from a ⟨*seq var*⟩ into the ⟨*tl var*⟩. Both of the variables are assigned locally. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker `\q_no_value`.

\seq_gpop:NN
\seq_gpop:cN

Updated: 2012-05-14

\seq_gpop:NN ⟨*seq var*⟩ ⟨*tl var*⟩

Pops the top item from a ⟨*seq var*⟩ into the ⟨*tl var*⟩. The ⟨*seq var*⟩ is modified globally, while the ⟨*tl var*⟩ is assigned locally. If ⟨*seq var*⟩ is empty the ⟨*tl var*⟩ is set to the special marker `\q_no_value`.

\seq_get:NN*TF*
\seq_get:cN*TF*

New: 2012-05-14
Updated: 2012-05-19

\seq_get:NNTF ⟨*seq var*⟩ ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*seq var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*seq var*⟩ is non-empty, stores the top item from a ⟨*seq var*⟩ in the ⟨*tl var*⟩ without removing it from the ⟨*seq var*⟩. The ⟨*tl var*⟩ is assigned locally.

\seq_pop:NN*TF*
\seq_pop:cN*TF*

New: 2012-05-14
Updated: 2012-05-19

\seq_pop:NNTF ⟨*seq var*⟩ ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*seq var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*seq var*⟩ is non-empty, pops the top item from the ⟨*seq var*⟩ in the ⟨*tl var*⟩, *i.e.* removes the item from the ⟨*seq var*⟩. Both the ⟨*seq var*⟩ and the ⟨*tl var*⟩ are assigned locally.

\seq_gpop:NN*TF*
\seq_gpop:cN*TF*

New: 2012-05-14
Updated: 2012-05-19

\seq_gpop:NNTF ⟨*seq var*⟩ ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*seq var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*seq var*⟩ is non-empty, pops the top item from the ⟨*seq var*⟩ in the ⟨*tl var*⟩, *i.e.* removes the item from the ⟨*seq var*⟩. The ⟨*seq var*⟩ is modified globally, while the ⟨*tl var*⟩ is assigned locally.

\seq_push:Nn
\seq_push:(NV|Nv|Ne|No|cn|cV|cv|ce|co)
\seq_gpush:Nn
\seq_gpush:(NV|Nv|Ne|No|cn|cV|cv|ce|co)

\seq_push:Nn ⟨*seq var*⟩ {⟨*item*⟩}

Adds the {⟨*item*⟩} to the top of the ⟨*seq var*⟩.

20.10 Sequences as sets

Sequences can also be used as sets, such that all of their items are distinct. Usage of sequences as sets is not currently widespread, hence no specific set function is provided. Instead, it is explained here how common set operations can be performed by combining several functions described in earlier sections. When using sequences to implement sets, one should be careful not to rely on the order of items in the sequence representing the set.

Sets should not contain several occurrences of a given item. To make sure that a ⟨*seq var*⟩ only has distinct items, use \seq_remove_duplicates:N ⟨*seq var*⟩. This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set ⟨*seq var*⟩ are straightforward. For instance, \seq_count:N ⟨*seq var*⟩ expands to the number of items, while \seq_if_in:NnTF ⟨*seq var*⟩ {⟨*item*⟩} tests if the ⟨*item*⟩ is in the set.

Adding an ⟨*item*⟩ to a set ⟨*seq var*⟩ can be done by appending it to the ⟨*seq var*⟩ if it is not already in the ⟨*seq var*⟩:

```
\seq_if_in:NnF ⟨seq var⟩ {⟨item⟩}
  { \seq_put_right:Nn ⟨seq var⟩ {⟨item⟩} }
```

Removing an ⟨*item*⟩ from a set ⟨*seq var*⟩ can be done using \seq_remove_all:Nn,

```
\seq_remove_all:Nn ⟨seq var⟩ {⟨item⟩}
```

The intersection of two sets ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ can be stored into ⟨*seq var*$_3$⟩ by collecting items of ⟨*seq var*$_1$⟩ which are in ⟨*seq var*$_2$⟩.

```
\seq_clear:N ⟨seq var3⟩
\seq_map_inline:Nn ⟨seq var1⟩
  {
    \seq_if_in:NnT ⟨seq var2⟩ {#1}
      { \seq_put_right:Nn ⟨seq var3⟩ {#1} }
  }
```

The code as written here only works if ⟨*seq var*$_3$⟩ is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence `\l__`⟨*pkg*⟩`_internal_seq`, then ⟨*seq var*$_3$⟩ should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ can be stored into ⟨*seq var*$_3$⟩ through

```
\seq_concat:NNN ⟨seq var3⟩ ⟨seq var1⟩ ⟨seq var2⟩
\seq_remove_duplicates:N ⟨seq var3⟩
```

or by adding items to (a copy of) ⟨*seq var*$_1$⟩ one by one

```
\seq_set_eq:NN ⟨seq var3⟩ ⟨seq var1⟩
\seq_map_inline:Nn ⟨seq var2⟩
  {
    \seq_if_in:NnF ⟨seq var3⟩ {#1}
      { \seq_put_right:Nn ⟨seq var3⟩ {#1} }
  }
```

The second approach is faster than the first when the ⟨*seq var*$_2$⟩ is short compared to ⟨*seq var*$_1$⟩.

The difference of two sets ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ can be stored into ⟨*seq var*$_3$⟩ by removing items of the ⟨*seq var*$_2$⟩ from (a copy of) the ⟨*seq var*$_1$⟩ one by one.

```
\seq_set_eq:NN ⟨seq var3⟩ ⟨seq var1⟩
\seq_map_inline:Nn ⟨seq var2⟩
  { \seq_remove_all:Nn ⟨seq var3⟩ {#1} }
```

The symmetric difference of two sets ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ can be stored into ⟨*seq var*$_3$⟩ by computing the difference between ⟨*seq var*$_1$⟩ and ⟨*seq var*$_2$⟩ and storing the result as `\l__`⟨*pkg*⟩`_internal_seq`, then the difference between ⟨*seq var*$_2$⟩ and ⟨*seq var*$_1$⟩, and finally concatenating the two differences to get the symmetric differences.

```
\seq_set_eq:NN \l__⟨pkg⟩_internal_seq ⟨seq var1⟩
\seq_map_inline:Nn ⟨seq var2⟩
  { \seq_remove_all:Nn \l__⟨pkg⟩_internal_seq {#1} }
\seq_set_eq:NN ⟨seq var3⟩ ⟨seq var2⟩
\seq_map_inline:Nn ⟨seq var1⟩
  { \seq_remove_all:Nn ⟨seq var3⟩ {#1} }
\seq_concat:NNN ⟨seq var3⟩ ⟨seq var3⟩ \l__⟨pkg⟩_internal_seq
```

20.11 Constant and scratch sequences

`\c_empty_seq` Constant that is always empty.

New: 2012-07-02

\l_tmpa_seq
\l_tmpb_seq

New: 2012-04-26

Scratch sequences for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_seq
\g_tmpb_seq

New: 2012-04-26

Scratch sequences for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

20.12 Viewing sequences

\seq_show:N
\seq_show:c

Updated: 2021-04-29

\seq_show:N ⟨*seq var*⟩

Displays the entries in the ⟨*seq var*⟩ in the terminal.

\seq_log:N
\seq_log:c

New: 2014-08-12
Updated: 2021-04-29

\seq_log:N ⟨*seq var*⟩

Writes the entries in the ⟨*seq var*⟩ in the log file.

Chapter 21

The l3int module
Integers

Calculation and comparison of integer values can be carried out using literal numbers, `int` registers, constants and integers stored in token list variables. The standard operators `+`, `-`, `/` and `*` and parentheses can be used within such expressions to carry arithmetic operations. This module carries out these functions on *integer expressions* ("⟨*int expr*⟩").

21.1 Integer expressions

Throughout this module, (almost) all `n`-type argument allow for an ⟨*intexpr*⟩ argument with the following syntax. The ⟨*integer expression*⟩ should consist, after expansion, of `+`, `-`, `*`, `/`, `(`, `)` and of course integer operands. The result is calculated by applying standard mathematical rules with the following peculiarities:

- `/` denotes division rounded to the closest integer with ties rounded away from zero;
- there is an error and the overall expression evaluates to zero whenever the absolute value of any intermediate result exceeds $2^{31}-1$, except in the case of scaling operations a`*`b`/`c, for which a`*`b may be arbitrarily large (but the operands a, b, c are still constrained to an absolute value at most $2^{31}-1$);
- parentheses may not appear after unary `+` or `-`, namely placing `+(` or `-(` at the start of an expression or after `+`, `-`, `*`, `/` or `(` leads to an error.

Each integer operand can be either an integer variable (with no need for `\int_use:N`) or an integer denotation. For example both

```
\int_show:n { 5 +  4 * 3 - ( 3 + 4 * 5 ) }
```

and

```
\tl_new:N  \l_my_tl
\tl_set:Nn \l_my_tl { 5 }
\int_new:N  \l_my_int
\int_set:Nn \l_my_int { 4 }
\int_show:n { \l_my_tl +  \l_my_int * 3 - ( 3 + 4 * 5 ) }
```

show the same result -6 because `\l_my_tl` expands to the integer denotation `5` while the integer variable `\l_my_int` takes the value 4. As the ⟨*integer expression*⟩ is fully expanded from left to right during evaluation, fully expandable and restricted-expandable functions can both be used, and `\exp_not:n` and its variants have no effect while `\exp_-not:N` may incorrectly interrupt the expression.

`\int_eval:n` ⋆

`\int_eval:n {⟨int expr⟩}`

Evaluates the ⟨*int expr*⟩ and leaves the result in the input stream as an integer denotation: for positive results an explicit sequence of decimal digits not starting with `0`, for negative results `-` followed by such a sequence, and `0` for zero.

TeXhackers note: Exactly two expansions are needed to evaluate `\int_eval:n`. The result is *not* an ⟨*internal integer*⟩, and therefore requires suitable termination if used in a TeX-style integer assignment.

As all TeX integers, integer operands can also be dimension or skip variables, converted to integers in `sp`, or octal numbers given as `'` followed by digits other than `8` and `9`, or hexadecimal numbers given as `"` followed by digits or upper case letters from `A` to `F`, or the character code of some character or one-character control sequence, given as `` ` ``⟨*char*⟩.

`\int_eval:w` ⋆

New: 2018-03-30

`\int_eval:w ⟨int expr⟩`

Evaluates the ⟨*int expr*⟩ as described for `\int_eval:n`. The end of the expression is the first token encountered that cannot form part of such an expression. If that token is `\scan_stop:` it is removed, otherwise not. Spaces do *not* terminate the expression. However, spaces terminate explicit integers, and this may terminate the expression: for instance, `\int_eval:w 1␣+␣1␣9` (with explicit space tokens inserted using `~` in a code setting) expands to `29` since the digit `9` is not part of the expression. Expansion details, etc., are as given for `\int_eval:n`.

`\int_sign:n` ⋆

New: 2018-11-03

`\int_sign:n {⟨int expr⟩}`

Evaluates the ⟨*int expr*⟩ then leaves 1 or 0 or -1 in the input stream according to the sign of the result.

`\int_abs:n` ⋆

Updated: 2012-09-26

`\int_abs:n {⟨int expr⟩}`

Evaluates the ⟨*int expr*⟩ as described for `\int_eval:n` and leaves the absolute value of the result in the input stream as an ⟨*integer denotation*⟩ after two expansions.

`\int_div_round:nn` ⋆

Updated: 2012-09-26

`\int_div_round:nn {⟨int expr_1⟩} {⟨int expr_2⟩}`

Evaluates the two ⟨*int expr*⟩s as described earlier, then divides the first value by the second, and rounds the result to the closest integer. Ties are rounded away from zero. Note that this is identical to using `/` directly in an ⟨*int expr*⟩. The result is left in the input stream as an ⟨*integer denotation*⟩ after two expansions.

`\int_div_truncate:nn` ⋆

Updated: 2012-02-09

`\int_div_truncate:nn {⟨int expr_1⟩} {⟨int expr_2⟩}`

Evaluates the two ⟨*int expr*⟩s as described earlier, then divides the first value by the second, and rounds the result towards zero. Note that division using `/` rounds to the closest integer instead. The result is left in the input stream as an ⟨*integer denotation*⟩ after two expansions.

`\int_max:nn` ⋆ `\int_min:nn` ⋆ Updated: 2012-09-26	`\int_max:nn {`⟨*int expr*$_1$⟩`} {`⟨*int expr*$_2$⟩`}` `\int_min:nn {`⟨*int expr*$_1$⟩`} {`⟨*int expr*$_2$⟩`}`

Evaluates the ⟨*int expr*⟩s as described for `\int_eval:n` and leaves either the larger or smaller value in the input stream as an ⟨*integer denotation*⟩ after two expansions.

`\int_mod:nn` ⋆ `\int_mod:nn {`⟨*int expr*$_1$⟩`} {`⟨*int expr*$_2$⟩`}`

Updated: 2012-09-26

Evaluates the two ⟨*int expr*⟩s as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is obtained by subtracting `\int_div_truncate:nn {`⟨*int expr*$_1$⟩`} {`⟨*int expr*$_2$⟩`}` times ⟨*int expr*$_2$⟩ from ⟨*int expr*$_1$⟩. Thus, the result has the same sign as ⟨*int expr*$_1$⟩ and its absolute value is strictly less than that of ⟨*int expr*$_2$⟩. The result is left in the input stream as an ⟨*integer denotation*⟩ after two expansions.

21.2 Creating and initialising integers

`\int_new:N`
`\int_new:c`

`\int_new:N` ⟨*integer*⟩

Creates a new ⟨*integer*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*integer*⟩ is initially equal to 0.

`\int_const:Nn`
`\int_const:cn`

Updated: 2011-10-22

`\int_const:Nn` ⟨*integer*⟩ `{`⟨*int expr*⟩`}`

Creates a new constant ⟨*integer*⟩ or raises an error if the name is already taken. The value of the ⟨*integer*⟩ is set globally to the ⟨*int expr*⟩.

`\int_zero:N`
`\int_zero:c`
`\int_gzero:N`
`\int_gzero:c`

`\int_zero:N` ⟨*integer*⟩

Sets ⟨*integer*⟩ to 0.

`\int_zero_new:N`
`\int_zero_new:c`
`\int_gzero_new:N`
`\int_gzero_new:c`

New: 2011-12-13

`\int_zero_new:N` ⟨*integer*⟩

Ensures that the ⟨*integer*⟩ exists globally by applying `\int_new:N` if necessary, then applies `\int_(g)zero:N` to leave the ⟨*integer*⟩ set to zero.

`\int_set_eq:NN`
`\int_set_eq:(cN|Nc|cc)`
`\int_gset_eq:NN`
`\int_gset_eq:(cN|Nc|cc)`

`\int_set_eq:NN` ⟨*integer*$_1$⟩ ⟨*integer*$_2$⟩

Sets the content of ⟨*integer*$_1$⟩ equal to that of ⟨*integer*$_2$⟩.

`\int_if_exist_p:N` ⋆
`\int_if_exist_p:c` ⋆
`\int_if_exist:N`*TF* ⋆
`\int_if_exist:c`*TF* ⋆

New: 2012-03-03

`\int_if_exist_p:N` ⟨*integer*⟩
`\int_if_exist:NTF` ⟨*integer*⟩ `{`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Tests whether the ⟨*integer*⟩ is currently defined. This does not check that the ⟨*integer*⟩ really is an integer variable.

21.3 Setting and incrementing integers

\int_add:Nn
\int_add:cn
\int_gadd:Nn
\int_gadd:cn

Updated: 2011-10-22

\int_add:Nn ⟨*integer*⟩ {⟨*int expr*⟩}

Adds the result of the ⟨*int expr*⟩ to the current content of the ⟨*integer*⟩.

\int_decr:N
\int_decr:c
\int_gdecr:N
\int_gdecr:c

\int_decr:N ⟨*integer*⟩

Decreases the value stored in ⟨*integer*⟩ by 1.

\int_incr:N
\int_incr:c
\int_gincr:N
\int_gincr:c

\int_incr:N ⟨*integer*⟩

Increases the value stored in ⟨*integer*⟩ by 1.

\int_set:Nn
\int_set:(cn|NV|cV)
\int_gset:Nn
\int_gset:(cn|cV)
\int_gset:MV

Updated: 2011-10-22

\int_set:Nn ⟨*integer*⟩ {⟨*int expr*⟩}

Sets ⟨*integer*⟩ to the value of ⟨*int expr*⟩, which must evaluate to an integer (as described for \int_eval:n).

\int_sub:Nn
\int_sub:cn
\int_gsub:Nn
\int_gsub:cn

Updated: 2011-10-22

\int_sub:Nn ⟨*integer*⟩ {⟨*int expr*⟩}

Subtracts the result of the ⟨*int expr*⟩ from the current content of the ⟨*integer*⟩.

21.4 Using integers

\int_use:N ⋆
\int_use:c ⋆

Updated: 2011-10-22

\int_use:N ⟨*integer*⟩

Recovers the content of an ⟨*integer*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where an ⟨*integer*⟩ is required (such as in the first and third arguments of \int_compare:nNnTF).

TEXhackers note: \int_use:N is the TEX primitive \the: this is one of several LATEX3 names for this primitive.

21.5 Integer expression conditionals

\int_compare_p:nNn ⋆
\int_compare:nNnTF ⋆

```
\int_compare_p:nNn {⟨int expr1⟩} ⟨relation⟩ {⟨int expr2⟩}
\int_compare:nNnTF
  {⟨int expr1⟩} ⟨relation⟩ {⟨int expr2⟩}
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates each of the ⟨*int expr*⟩s as described for `\int_eval:n`. The two results are then compared using the ⟨*relation*⟩:

Equal	=
Greater than	>
Less than	<

This function is less flexible than `\int_compare:nTF` but around 5 times faster.

\int_compare_p:n ⋆
\int_compare:nTF ⋆

Updated: 2013-01-13

```
\int_compare_p:n
  {
    ⟨int expr1⟩ ⟨relation1⟩
    ...
    ⟨int exprN⟩ ⟨relationN⟩
    ⟨int exprN+1⟩
  }
\int_compare:nTF
  {
    ⟨int expr1⟩ ⟨relation1⟩
    ...
    ⟨int exprN⟩ ⟨relationN⟩
    ⟨int exprN+1⟩
  }
  {⟨true code⟩} {⟨false code⟩}
```

This function evaluates the ⟨*int expr*⟩s as described for `\int_eval:n` and compares consecutive result using the corresponding ⟨*relation*⟩, namely it compares ⟨*int expr*$_1$⟩ and ⟨*int expr*$_2$⟩ using the ⟨*relation*$_1$⟩, then ⟨*int expr*$_2$⟩ and ⟨*int expr*$_3$⟩ using the ⟨*relation*$_2$⟩, until finally comparing ⟨*int expr*$_N$⟩ and ⟨*int expr*$_{N+1}$⟩ using the ⟨*relation*$_N$⟩. The test yields `true` if all comparisons are `true`. Each ⟨*int expr*⟩ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other ⟨*integer expression*⟩ is evaluated and no other comparison is performed. The ⟨*relations*⟩ can be any of the following:

Equal	= or ==
Greater than or equal to	>=
Greater than	>
Less than or equal to	<=
Less than	<
Not equal	!=

This function is more flexible than `\int_compare:nNnTF` but around 5 times slower.

\int_case:nn ⋆
\int_case:nnTF ⋆

New: 2013-07-24

```
\int_case:nnTF {⟨test int expr⟩}
  {
    {⟨int expr case₁⟩} {⟨code case₁⟩}
    {⟨int expr case₂⟩} {⟨code case₂⟩}
    ...
    {⟨int expr caseₙ⟩} {⟨code caseₙ⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function evaluates the ⟨*test int expr*⟩ and compares this in turn to each of the ⟨*int expr case*⟩s until a match is found. If the two are equal then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\int_case:nn`, which does nothing if there is no match, is also available. For example

```
\int_case:nnF
  { 2 * 5 }
  {
    { 5 }       { Small }
    { 4 + 6 }   { Medium }
    { -2 * 10 } { Negative }
  }
  { No idea! }
```

leaves "`Medium`" in the input stream. Since evaluation of the test expressions stops at the first successful case, the order of possible matches should normally be that the most likely are earlier: this will reduce the average steps required to complete expansion.

\int_if_even_p:n ⋆
\int_if_even:nTF ⋆
\int_if_odd_p:n ⋆
\int_if_odd:nTF ⋆

```
\int_if_odd_p:n {⟨int expr⟩}
\int_if_odd:nTF {⟨int expr⟩}
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates the ⟨*int expr*⟩ as described for `\int_eval:n`. It then evaluates if this is odd or even, as appropriate.

\int_if_zero_p:n ⋆
\int_if_zero:nTF ⋆

New: 2023-05-17

```
\int_if_zero_p:n {⟨int expr⟩}
\int_if_zero:nTF {⟨int expr⟩}
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates the ⟨*int expr*⟩ as described for `\int_eval:n`. It then evaluates if this is zero or not.

21.6 Integer expression loops

\int_do_until:nNnn ☆

```
\int_do_until:nNnn {⟨int expr₁⟩} ⟨relation⟩ {⟨int expr₂⟩} {⟨code⟩}
```

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*int expr*⟩s as described for `\int_compare:nNnTF`. If the test is `false` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `true`.

`\int_do_while:nNnn` ☆ `\int_do_while:nNnn {⟨int expr_1⟩} ⟨relation⟩ {⟨int expr_2⟩} {⟨code⟩}`

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the relationship between the two ⟨*int expr*⟩s as described for `\int_compare:nNnTF`. If the test is `true` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `false`.

`\int_until_do:nNnn` ☆ `\int_until_do:nNnn {⟨int expr_1⟩} ⟨relation⟩ {⟨int expr_2⟩} {⟨code⟩}`

Evaluates the relationship between the two ⟨*int expr*⟩s as described for `\int_compare:nNnTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `false`. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is `true`.

`\int_while_do:nNnn` ☆ `\int_while_do:nNnn {⟨int expr_1⟩} ⟨relation⟩ {⟨int expr_2⟩} {⟨code⟩}`

Evaluates the relationship between the two ⟨*int expr*⟩s as described for `\int_compare:nNnTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `true`. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is `false`.

`\int_do_until:nn` ☆ `\int_do_until:nn {⟨integer relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the ⟨*integer relation*⟩ as described for `\int_compare:nTF`. If the test is `false` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `true`.

`\int_do_while:nn` ☆ `\int_do_while:nn {⟨integer relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the ⟨*integer relation*⟩ as described for `\int_compare:nTF`. If the test is `true` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `false`.

`\int_until_do:nn` ☆ `\int_until_do:nn {⟨integer relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Evaluates the ⟨*integer relation*⟩ as described for `\int_compare:nTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `false`. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is `true`.

`\int_while_do:nn` ☆ `\int_while_do:nn {⟨integer relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Evaluates the ⟨*integer relation*⟩ as described for `\int_compare:nTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `true`. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is `false`.

21.7 Integer step functions

\int_step_function:nN ☆
\int_step_function:nnN ☆
\int_step_function:nnnN ☆

New: 2012-06-04
Updated: 2018-04-22

\int_step_function:nN {⟨*final value*⟩} ⟨*function*⟩
\int_step_function:nnN {⟨*initial value*⟩} {⟨*final value*⟩} ⟨*function*⟩
\int_step_function:nnnN {⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} ⟨*function*⟩

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be integer expressions. The ⟨*function*⟩ is then placed in front of each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩). The ⟨*step*⟩ must be non-zero. If the ⟨*step*⟩ is positive, the loop stops when the ⟨*value*⟩ becomes larger than the ⟨*final value*⟩. If the ⟨*step*⟩ is negative, the loop stops when the ⟨*value*⟩ becomes smaller than the ⟨*final value*⟩. The ⟨*function*⟩ should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\int_step_function:nnnN { 1 } { 1 } { 5 } \my_func:n
```

would print

[I saw 1] [I saw 2] [I saw 3] [I saw 4] [I saw 5]

The functions \int_step_function:nN and \int_step_function:nnN both use a fixed ⟨*step*⟩ of 1, and in the case of \int_step_function:nN the ⟨*initial value*⟩ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

\int_step_inline:nn
\int_step_inline:nnn
\int_step_inline:nnnn

New: 2012-06-04
Updated: 2018-04-22

\int_step_inline:nn {⟨*final value*⟩} {⟨*code*⟩}
\int_step_inline:nnn {⟨*initial value*⟩} {⟨*final value*⟩} {⟨*code*⟩}
\int_step_inline:nnnn {⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} {⟨*code*⟩}

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be integer expressions. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream with #1 replaced by the current ⟨*value*⟩. Thus the ⟨*code*⟩ should define a function of one argument (#1).

The functions \int_step_inline:nn and \int_step_inline:nnn both use a fixed ⟨*step*⟩ of 1, and in the case of \int_step_inline:nn the ⟨*initial value*⟩ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

\int_step_variable:nNn
\int_step_variable:nnNn
\int_step_variable:nnnNn

New: 2012-06-04
Updated: 2018-04-22

\int_step_variable:nNn {⟨*final value*⟩} ⟨*tl var*⟩ {⟨*code*⟩}
\int_step_variable:nnNn {⟨*initial value*⟩} {⟨*final value*⟩} ⟨*tl var*⟩ {⟨*code*⟩}
\int_step_variable:nnnNn {⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} ⟨*tl var*⟩ {⟨*code*⟩}

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be integer expressions. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream, with the ⟨*tl var*⟩ defined as the current ⟨*value*⟩. Thus the ⟨*code*⟩ should make use of the ⟨*tl var*⟩.

The functions \int_step_variable:nNn and \int_step_variable:nnNn both use a fixed ⟨*step*⟩ of 1, and in the case of \int_step_variable:nNn the ⟨*initial value*⟩ is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

21.8 Formatting integers

Integers can be placed into the output stream with formatting. These conversions apply to any integer expressions.

\int_to_arabic:n ⋆
\int_to_arabic:V ⋆

Updated: 2011-10-22

\int_to_arabic:n {⟨*int expr*⟩}

Places the value of the ⟨*int expr*⟩ in the input stream as digits, with category code 12 (other).

\int_to_alph:n ⋆
\int_to_Alph:n ⋆

Updated: 2011-09-17

\int_to_alph:n {⟨*int expr*⟩}

Evaluates the ⟨*int expr*⟩ and converts the result into a series of letters, which are then left in the input stream. The conversion rule uses the 26 letters of the English alphabet, in order, adding letters when necessary to increase the total possible range of representable numbers. Thus

```
\int_to_alph:n { 1 }
```

places a in the input stream,

```
\int_to_alph:n { 26 }
```

is represented as z and

```
\int_to_alph:n { 27 }
```

is converted to aa. For conversions using other alphabets, use \int_to_symbols:nnn to define an alphabet-specific function. The basic \int_to_alph:n and \int_to_Alph:n functions should not be modified. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

\int_to_symbols:nnn ⋆

Updated: 2011-09-17

\int_to_symbols:nnn
  {⟨*int expr*⟩} {⟨*total symbols*⟩}
  {⟨*value to symbol mapping*⟩}

This is the low-level function for conversion of an ⟨*int expr*⟩ into a symbolic form (often letters). The ⟨*total symbols*⟩ available should be given as an integer expression. Values are actually converted to symbols according to the ⟨*value to symbol mapping*⟩. This should be given as ⟨*total symbols*⟩ pairs of entries, a number and the appropriate symbol. Thus the \int_to_alph:n function is defined as

```
\cs_new:Npn \int_to_alph:n #1
  {
    \int_to_symbols:nnn {#1} { 26 }
      {
        {  1 } { a }
        {  2 } { b }
        ...
        { 26 } { z }
      }
  }
```

`\int_to_bin:n` ⋆

New: 2014-02-11

`\int_to_bin:n {⟨int expr⟩}`

Calculates the value of the ⟨*int expr*⟩ and places the binary representation of the result in the input stream.

`\int_to_hex:n` ⋆
`\int_to_Hex:n` ⋆

New: 2014-02-11

`\int_to_hex:n {⟨int expr⟩}`

Calculates the value of the ⟨*int expr*⟩ and places the hexadecimal (base 16) representation of the result in the input stream. Letters are used for digits beyond 9: lower case letters for `\int_to_hex:n` and upper case ones for `\int_to_Hex:n`. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

`\int_to_oct:n` ⋆

New: 2014-02-11

`\int_to_oct:n {⟨int expr⟩}`

Calculates the value of the ⟨*int expr*⟩ and places the octal (base 8) representation of the result in the input stream. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

`\int_to_base:nn` ⋆
`\int_to_Base:nn` ⋆

Updated: 2014-02-11

`\int_to_base:nn {⟨int expr⟩} {⟨base⟩}`

Calculates the value of the ⟨*int expr*⟩ and converts it into the appropriate representation in the ⟨*base*⟩; the later may be given as an integer expression. For bases greater than 10 the higher "digits" are represented by letters from the English alphabet: lower case letters for `\int_to_base:n` and upper case ones for `\int_to_Base:n`. The maximum ⟨*base*⟩ value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

TeXhackers note: This is a generic version of `\int_to_bin:n`, *etc.*

`\int_to_roman:n` ☆
`\int_to_Roman:n` ☆

Updated: 2011-10-22

`\int_to_roman:n {⟨int expr⟩}`

Places the value of the ⟨*int expr*⟩ in the input stream as Roman numerals, either lower case (`\int_to_roman:n`) or upper case (`\int_to_Roman:n`). If the value is negative or zero, the output is empty. The Roman numerals are letters with category code 11 (letter). The letters used are `mdclxvi`, repeated as needed: the notation with bars (such as $\bar{\mathrm{v}}$ for 5000) is *not* used. For instance `\int_to_roman:n { 8249 }` expands to `mmmmmmmmccxlix`.

21.9 Converting from other formats to integers

`\int_from_alph:n` ⋆

Updated: 2014-08-25

`\int_from_alph:n {⟨letters⟩}`

Converts the ⟨*letters*⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨*letters*⟩ are first converted to a string, with no expansion. Lower and upper case letters from the English alphabet may be used, with "a" equal to 1 through to "z" equal to 26. The function also accepts a leading sign, made of `+` and `-`. This is the inverse function of `\int_to_alph:n` and `\int_to_Alph:n`.

`\int_from_bin:n` ⋆

New: 2014-02-11
Updated: 2014-08-25

`\int_from_bin:n {⟨binary number⟩}`

Converts the ⟨*binary number*⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨*binary number*⟩ is first converted to a string, with no expansion. The function accepts a leading sign, made of `+` and `-`, followed by binary digits. This is the inverse function of `\int_to_bin:n`.

\int_from_hex:n ⋆ `\int_from_hex:n {⟨hexadecimal number⟩}`

New: 2014-02-11
Updated: 2014-08-25

Converts the ⟨*hexadecimal number*⟩ into the integer (base 10) representation and leaves this in the input stream. Digits greater than 9 may be represented in the ⟨*hexadecimal number*⟩ by upper or lower case letters. The ⟨*hexadecimal number*⟩ is first converted to a string, with no expansion. The function also accepts a leading sign, made of `+` and `-`. This is the inverse function of `\int_to_hex:n` and `\int_to_Hex:n`.

\int_from_oct:n ⋆ `\int_from_oct:n {⟨octal number⟩}`

New: 2014-02-11
Updated: 2014-08-25

Converts the ⟨*octal number*⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨*octal number*⟩ is first converted to a string, with no expansion. The function accepts a leading sign, made of `+` and `-`, followed by octal digits. This is the inverse function of `\int_to_oct:n`.

\int_from_roman:n ⋆ `\int_from_roman:n {⟨roman numeral⟩}`

Updated: 2014-08-25

Converts the ⟨*roman numeral*⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨*roman numeral*⟩ is first converted to a string, with no expansion. The ⟨*roman numeral*⟩ may be in upper or lower case; if the numeral contains characters besides `mdclxvi` or `MDCLXVI` then the resulting value is -1. This is the inverse function of `\int_to_roman:n` and `\int_to_Roman:n`.

\int_from_base:nn ⋆ `\int_from_base:nn {⟨number⟩} {⟨base⟩}`

Updated: 2014-08-25

Converts the ⟨*number*⟩ expressed in ⟨*base*⟩ into the appropriate value in base 10. The ⟨*number*⟩ is first converted to a string, with no expansion. The ⟨*number*⟩ should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum ⟨*base*⟩ value is 36. This is the inverse function of `\int_to_base:nn` and `\int_to_Base:nn`.

21.10 Random integers

\int_rand:nn ⋆ `\int_rand:nn {⟨int expr₁⟩} {⟨int expr₂⟩}`

New: 2016-12-06
Updated: 2018-04-27

Evaluates the two ⟨*int expr*⟩s and produces a pseudo-random number between the two (with bounds included).

\int_rand:n ⋆ `\int_rand:n {⟨int expr⟩}`

New: 2018-05-05

Evaluates the ⟨*int expr*⟩ then produces a pseudo-random number between 1 and the ⟨*int expr*⟩ (included).

21.11 Viewing integers

\int_show:N
\int_show:c

`\int_show:N ⟨integer⟩`

Displays the value of the ⟨*integer*⟩ on the terminal.

`\int_show:n`

New: 2011-11-22
Updated: 2015-08-07

`\int_show:n {⟨int expr⟩}`

Displays the result of evaluating the ⟨*int expr*⟩ on the terminal.

`\int_log:N`
`\int_log:c`

New: 2014-08-22
Updated: 2015-08-03

`\int_log:N ⟨integer⟩`

Writes the value of the ⟨*integer*⟩ in the log file.

`\int_log:n`

New: 2014-08-22
Updated: 2015-08-07

`\int_log:n {⟨int expr⟩}`

Writes the result of evaluating the ⟨*int expr*⟩ in the log file.

21.12 Constant integers

`\c_zero_int`
`\c_one_int`

New: 2018-05-07

Integer values used with primitive tests and assignments: their self-terminating nature makes these more convenient and faster than literal numbers.

`\c_max_int` The maximum value that can be stored as an integer.

`\c_max_register_int` Maximum number of registers.

`\c_max_char_int` Maximum character code completely supported by the engine.

21.13 Scratch integers

`\l_tmpa_int`
`\l_tmpb_int`

Scratch integer for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_int`
`\g_tmpb_int`

Scratch integer for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

21.14 Direct number expansion

\int_value:w ⋆

New: 2018-03-27

```
\int_value:w ⟨integer⟩
\int_value:w ⟨integer denotation⟩ ⟨optional space⟩
```

Expands the following tokens until an ⟨*integer*⟩ is formed, and leaves a normalized form (no leading sign except for negative numbers, no leading digit `0` except for zero) in the input stream as category code 12 (other) characters. The ⟨*integer*⟩ can consist of any number of signs (with intervening spaces) followed by

- an integer variable (in fact, any TEX register except `\toks`) or
- explicit digits (or by `'`⟨*octal digits*⟩ or `"`⟨*hexadecimal digits*⟩ or `` ` ``⟨*character*⟩).

In this last case expansion stops once a non-digit is found; if that is a space it is removed as in `f`-expansion, and so `\exp_stop_f:` may be employed as an end marker. Note that protected functions *are* expanded by this process.

This function requires exactly one expansion to produce a value, and so is suitable for use in cases where a number is required "directly". In general, `\int_eval:n` is the preferred approach to generating numbers.

TEXhackers note: This is the TEX primitive `\number`.

21.15 Primitive conditionals

\if_int_compare:w ⋆

```
\if_int_compare:w ⟨integer1⟩ ⟨relation⟩ ⟨integer2⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Compare two integers using ⟨*relation*⟩, which must be one of `=`, `<` or `>` with category code 12. The `\else:` branch is optional.

TEXhackers note: This is the TEX primitive `\ifnum`.

\if_case:w ⋆
\or: ⋆

```
\if_case:w ⟨integer⟩ ⟨case0⟩
  \or: ⟨case1⟩
  \or: ...
  \else: ⟨default⟩
\fi:
```

Selects a case to execute based on the value of the ⟨*integer*⟩. The first case (⟨$case_0$⟩) is executed if ⟨*integer*⟩ is 0, the second (⟨$case_1$⟩) if the ⟨*integer*⟩ is 1, *etc.* The ⟨*integer*⟩ may be a literal, a constant or an integer expression (*e.g.* using `\int_eval:n`).

TEXhackers note: These are the TEX primitives `\ifcase` and `\or`.

`\if_int_odd:w` ⋆

```
\if_int_odd:w ⟨tokens⟩ ⟨optional space⟩
  ⟨true code⟩
\else:
  ⟨true code⟩
\fi:
```

Expands ⟨*tokens*⟩ until a non-numeric token or a space is found, and tests whether the resulting ⟨*integer*⟩ is odd. If so, ⟨*true code*⟩ is executed. The `\else:` branch is optional.

> **TEXhackers note:** This is the TEX primitive `\ifodd`.

Chapter 22

The l3flag module
Expandable flags

Flags are the only data-type that can be modified in expansion-only contexts. This module is meant mostly for kernel use: in almost all cases, booleans or integers should be preferred to flags because they are very significantly faster.

A flag can hold any (small) non-negative value, which we call its ⟨*height*⟩. In expansion-only contexts, a flag can only be "raised": this increases the ⟨*height*⟩ by 1. The ⟨*height*⟩ can also be queried expandably. However, decreasing it, or setting it to zero requires non-expandable assignments.

Flag variables are always local.

A typical use case of flags would be to keep track of whether an exceptional condition has occurred during expandable processing, and produce a meaningful (non-expandable) message after the end of the expandable processing. This is exemplified by l3str-convert, which for performance reasons performs conversions of individual characters expandably and for readability reasons produces a single error message describing incorrect inputs that were encountered.

Flags should not be used without carefully considering the fact that raising a flag takes a time and memory proportional to its height and that the memory cannot be reclaimed even if the flag is cleared. Flags should not be used unless it is unavoidable.

In earlier versions, flags were referenced by an `n`-type ⟨*flag name*⟩ such as `fp_overflow`, used as part of `\use:c` constructions. All of the commands described below have `n`-type analogues that can still appear in old code, but the `N`-type commands are to be preferred moving forward. The `n`-type ⟨*flag name*⟩ is simply mapped to `\l_`⟨*flag name*⟩`_flag`, which makes it easier for packages using public flags (such as l3fp) to retain backwards compatibility.

22.1 Setting up flags

`\flag_new:N`
`\flag_new:c`
New: 2024-01-12

`\flag_new:N` ⟨*flag var*⟩

Creates a new ⟨*flag var*⟩, or raises an error if the name is already taken. The declaration is global, but flags are always local variables. The ⟨*flag var*⟩ initially has zero height.

\flag_clear:N
\flag_clear:c

New: 2024-01-12

\flag_clear:N ⟨*flag var*⟩

Sets the height of the ⟨*flag var*⟩ to zero. The assignment is local.

\flag_clear_new:N
\flag_clear_new:c

New: 2024-01-12

\flag_clear_new:N ⟨*flag var*⟩

Ensures that the ⟨*flag var*⟩ exists globally by applying \flag_new:N if necessary, then applies \flag_clear:N, setting the height to zero locally.

\flag_show:N
\flag_show:c

New: 2024-01-12

\flag_show:N ⟨*flag var*⟩

Displays the height of the ⟨*flag var*⟩ in the terminal.

\flag_log:N
\flag_log:c

New: 2024-01-12

\flag_log:N ⟨*flag var*⟩

Writes the height of the ⟨*flag var*⟩ in the log file.

22.2 Expandable flag commands

\flag_if_exist_p:N ⋆
\flag_if_exist_p:c ⋆
\flag_if_exist:N*TF* ⋆
\flag_if_exist:c*TF* ⋆

New: 2024-01-12

\flag_if_exist_p:N ⟨*flag var*⟩
\flag_if_exist:NTF ⟨*flag var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

This function returns true if the ⟨*flag var*⟩ is currently defined, and false otherwise. This does not check that the ⟨*flag var*⟩ really is a flag variable.

\flag_if_raised_p:N ⋆
\flag_if_raised_p:c ⋆
\flag_if_raised:N*TF* ⋆
\flag_if_raised:c*TF* ⋆

New: 2024-01-12

\flag_if_raised_p:N ⟨*flag var*⟩
\flag_if_raised:NTF ⟨*flag var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

This function returns true if the ⟨*flag var*⟩ has non-zero height, and false if the ⟨*flag var*⟩ has zero height.

\flag_height:N ⋆
\flag_height:c ⋆

New: 2024-01-12

\flag_height:N ⟨*flag var*⟩

Expands to the height of the ⟨*flag var*⟩ as an integer denotation.

\flag_raise:N ⋆
\flag_raise:c ⋆

New: 2024-01-12

\flag_raise:N ⟨*flag var*⟩

The height of ⟨*flag var*⟩ is increased by 1 locally.

\flag_ensure_raised:N ⋆
\flag_ensure_raised:c ⋆

New: 2024-01-12

\flag_ensure_raised:N ⟨*flag var*⟩

Ensures the ⟨*flag var*⟩ is raised by making its height at least 1, locally.

`\l_tmpa_flag`
`\l_tmpb_flag`

New: 2024-01-12

Scratch flag for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Chapter 23

The l3clist module Comma separated lists

Comma lists (in short, `clist`) contain ordered data where items can be added to the left or right end of the list. This data type allows basic list manipulations such as adding/removing items, applying a function to every item, removing duplicate items, extracting a given item, using the comma list with specified separators, and so on. Sequences (defined in l3seq) are safer, faster, and provide more features, so they should often be preferred to comma lists. Comma lists are mostly useful when interfacing with LaTeX 2_ε or other code that expects or provides items separated by commas.

Several items can be added at once. To ease input of comma lists from data provided by a user outside an `\ExplSyntaxOn` … `\ExplSyntaxOff` block, spaces are removed from both sides of each comma-delimited argument upon input. Blank arguments are ignored, to allow for trailing commas or repeated commas (which may otherwise arise when concatenating comma lists "by hand"). In addition, a set of braces is removed if the result of space-trimming is braced: this allows the storage of any item in a comma list. For instance,

```
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { ~a~ , ~{b}~ , c~\d }
\clist_put_right:Nn \l_my_clist { ~{e~} , , {{f}} , }
```

results in `\l_my_clist` containing `a,b,c~\d,{e~},{{f}}` namely the five items `a`, `b`, `c~\d`, `e~` and `{f}`. Comma lists normally do not contain empty or blank items so the following gives an empty comma list:

```
\clist_clear_new:N \l_my_clist
\clist_set:Nn \l_my_clist { , ~ , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
```

and it leaves `true` in the input stream. To include an "unsafe" item (empty, or one that contains a comma, or starts or ends with a space, or is a single brace group), surround it with braces.

Any `n`-type token list is a valid comma list input for l3clist functions, which will split the token list at every comma and process the items as described above. On the other hand, `N`-type functions expect comma list variables, which are particular token list variables in which this processing of items (and removal of blank items) has already occurred. Because comma list variables are token list variables, expanding them once yields their items separated by commas, and l3tl functions such as \tl_show:N can be applied to them. (These functions often have l3clist analogues, which should be preferred.)

Almost all operations on comma lists are noticeably slower than those on sequences so converting the data to sequences using \seq_set_from_clist:Nn (see l3seq) may be advisable if speed is important. The exception is that \clist_if_in:NnTF and \clist_remove_duplicates:N may be faster than their sequence analogues for large lists. However, these functions work slowly for "unsafe" items that must be braced, and may produce errors when their argument contains {, } or # (assuming the usual TeX category codes apply). The sequence data type should thus certainly be preferred to comma lists to store such items.

23.1 Creating and initialising comma lists

\clist_new:N
\clist_new:c

\clist_new:N ⟨*clist var*⟩

Creates a new ⟨*clist var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*clist var*⟩ initially contains no items.

\clist_const:Nn
\clist_const:(Ne|cn|ce)

New: 2014-07-05

\clist_const:Nn ⟨*clist var*⟩ {⟨*comma list*⟩}

Creates a new constant ⟨*clist var*⟩ or raises an error if the name is already taken. The value of the ⟨*clist var*⟩ is set globally to the ⟨*comma list*⟩.

\clist_clear:N
\clist_clear:c
\clist_gclear:N
\clist_gclear:c

\clist_clear:N ⟨*clist var*⟩

Clears all items from the ⟨*clist var*⟩.

\clist_clear_new:N
\clist_clear_new:c
\clist_gclear_new:N
\clist_gclear_new:c

\clist_clear_new:N ⟨*clist var*⟩

Ensures that the ⟨*clist var*⟩ exists globally by applying \clist_new:N if necessary, then applies \clist_(g)clear:N to leave the list empty.

\clist_set_eq:NN
\clist_set_eq:(cN|Nc|cc)
\clist_gset_eq:NN
\clist_gset_eq:(cN|Nc|cc)

\clist_set_eq:NN ⟨*clist var*$_1$⟩ ⟨*clist var*$_2$⟩

Sets the content of ⟨*clist var*$_1$⟩ equal to that of ⟨*clist var*$_2$⟩. To set a token list variable equal to a comma list variable, use \tl_set_eq:NN. Conversely, setting a comma list variable to a token list is unadvisable unless one checks space-trimming and related issues.

\clist_set_from_seq:NN
\clist_set_from_seq:(cN|Nc|cc)
\clist_gset_from_seq:NN
\clist_gset_from_seq:(cN|Nc|cc)

New: 2014-07-17

\clist_set_from_seq:NN ⟨*clist var*⟩ ⟨*seq var*⟩

Converts the data in the ⟨*seq var*⟩ into a ⟨*clist var*⟩: the original ⟨*seq var*⟩ is unchanged. Items which contain either spaces or commas are surrounded by braces.

`\clist_concat:NNN`
`\clist_concat:ccc`
`\clist_gconcat:NNN`
`\clist_gconcat:ccc`

`\clist_concat:NNN` ⟨*clist var*$_1$⟩ ⟨*clist var*$_2$⟩ ⟨*clist var*$_3$⟩

Concatenates the content of ⟨*clist var*$_2$⟩ and ⟨*clist var*$_3$⟩ together and saves the result in ⟨*clist var*$_1$⟩. The items in ⟨*clist var*$_2$⟩ are placed at the left side of the new comma list.

`\clist_if_exist_p:N` ⋆
`\clist_if_exist_p:c` ⋆
`\clist_if_exist:NTF` ⋆
`\clist_if_exist:cTF` ⋆

New: 2012-03-03

`\clist_if_exist_p:N` ⟨*clist var*⟩
`\clist_if_exist:NTF` ⟨*clist var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*clist var*⟩ is currently defined. This does not check that the ⟨*clist var*⟩ really is a comma list.

23.2 Adding data to comma lists

`\clist_set:Nn`
`\clist_set:(NV|Ne|No|cn|cV|ce|co)`
`\clist_gset:Nn`
`\clist_gset:(NV|Ne|No|cn|cV|ce|co)`

New: 2011-09-06

`\clist_set:Nn` ⟨*clist var*⟩ {⟨*item*$_1$⟩,...,⟨*item*$_n$⟩}

Sets ⟨*clist var*⟩ to contain the ⟨*items*⟩, removing any previous content from the variable. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To store some ⟨*tokens*⟩ as a single ⟨*item*⟩ even if the ⟨*tokens*⟩ contain commas or spaces, add a set of braces: `\clist_set:Nn` ⟨*clist var*⟩ { {⟨*tokens*⟩} }.

`\clist_put_left:Nn`
`\clist_put_left:(NV|Nv|Ne|No|cn|cV|cv|ce|co)`
`\clist_gput_left:Nn`
`\clist_gput_left:(NV|Nv|Ne|No|cn|cV|cv|ce|co)`

Updated: 2011-09-05

`\clist_put_left:Nn` ⟨*clist var*⟩ {⟨*item*$_1$⟩,...,⟨*item*$_n$⟩}

Appends the ⟨*items*⟩ to the left of the ⟨*clist var*⟩. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some ⟨*tokens*⟩ as a single ⟨*item*⟩ even if the ⟨*tokens*⟩ contain commas or spaces, add a set of braces: `\clist_put_left:Nn` ⟨*clist var*⟩ { {⟨*tokens*⟩} }.

`\clist_put_right:Nn`
`\clist_put_right:(NV|Nv|Ne|No|cn|cV|cv|ce|co)`
`\clist_gput_right:Nn`
`\clist_gput_right:(NV|Nv|Ne|No|cn|cV|cv|ce|co)`

Updated: 2011-09-05

`\clist_put_right:Nn` ⟨*clist var*⟩ {⟨*item*$_1$⟩,...,⟨*item*$_n$⟩}

Appends the ⟨*items*⟩ to the right of the ⟨*clist var*⟩. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some ⟨*tokens*⟩ as a single ⟨*item*⟩ even if the ⟨*tokens*⟩ contain commas or spaces, add a set of braces: `\clist_put_right:Nn` ⟨*clist var*⟩ { {⟨*tokens*⟩} }.

23.3 Modifying comma lists

While comma lists are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update comma lists, while retaining the order of the unaffected entries.

`\clist_remove_duplicates:N`
`\clist_remove_duplicates:c`
`\clist_gremove_duplicates:N`
`\clist_gremove_duplicates:c`

`\clist_remove_duplicates:N ⟨clist var⟩`

Removes duplicate items from the ⟨clist var⟩, leaving the left most copy of each item in the ⟨clist var⟩. The ⟨item⟩ comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

TeXhackers note: This function iterates through every item in the ⟨clist var⟩ and does a comparison with the ⟨items⟩ already checked. It is therefore relatively slow with large comma lists. Furthermore, it may fail if any of the items in the ⟨clist var⟩ contains `{`, `}`, or `#` (assuming the usual TeX category codes apply).

`\clist_remove_all:Nn`
`\clist_remove_all:(cn|NV|cV|Ne|ce)`
`\clist_gremove_all:Nn`
`\clist_gremove_all:(cn|NV|cV|Ne|ce)`

Updated: 2011-09-06

`\clist_remove_all:Nn ⟨clist var⟩ {⟨item⟩}`

Removes every occurrence of ⟨item⟩ from the ⟨clist var⟩. The ⟨item⟩ comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

TeXhackers note: The function may fail if the ⟨item⟩ contains `{`, `}`, or `#` (assuming the usual TeX category codes apply).

`\clist_reverse:N`
`\clist_reverse:c`
`\clist_greverse:N`
`\clist_greverse:c`

New: 2014-07-18

`\clist_reverse:N ⟨clist var⟩`

Reverses the order of items stored in the ⟨clist var⟩.

`\clist_reverse:n` ⋆

New: 2014-07-18

`\clist_reverse:n {⟨comma list⟩}`

Leaves the items in the ⟨comma list⟩ in the input stream in reverse order. Contrarily to other what is done for other `n`-type ⟨comma list⟩ arguments, braces and spaces are preserved by this process.

TeXhackers note: The result is returned within `\unexpanded`, which means that the comma list does not expand further when appearing in an `e`-type or `x`-type argument expansion.

\clist_sort:Nn
\clist_sort:cn
\clist_gsort:Nn
\clist_gsort:cn

New: 2017-02-06

\clist_sort:Nn ⟨*clist var*⟩ {⟨*comparison code*⟩}

Sorts the items in the ⟨*clist var*⟩ according to the ⟨*comparison code*⟩, and assigns the result to ⟨*clist var*⟩. The details of sorting comparison are described in Section 6.1.

23.4 Comma list conditionals

\clist_if_empty_p:N ⋆
\clist_if_empty_p:c ⋆
\clist_if_empty:N*TF* ⋆
\clist_if_empty:c*TF* ⋆

\clist_if_empty_p:N ⟨*clist var*⟩
\clist_if_empty:NTF ⟨*clist var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*clist var*⟩ is empty (containing no items).

\clist_if_empty_p:n ⋆
\clist_if_empty:n*TF* ⋆

New: 2014-07-05

\clist_if_empty_p:n {⟨*comma list*⟩}
\clist_if_empty:nTF {⟨*comma list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*comma list*⟩ is empty (containing no items). The rules for space trimming are as for other n-type comma-list functions, hence the comma list {~,~,,~} (without outer braces) is empty, while {~,{},} (without outer braces) contains one element, which happens to be empty: the comma-list is not empty.

\clist_if_in:Nn*TF*
\clist_if_in:(NV|No|cn|cV|co)*TF*
\clist_if_in:nn*TF*
\clist_if_in:(nV|no)*TF*

Updated: 2011-09-06

\clist_if_in:NnTF ⟨*clist var*⟩ {⟨*item*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*item*⟩ is present in the ⟨*clist var*⟩. In the case of an n-type ⟨*comma list*⟩, the usual rules of space trimming and brace stripping apply. Hence,

```
\clist_if_in:nnTF { a , {b}~ , {b} , c } { b } {true} {false}
```

yields true.

TEXhackers note: The function may fail if the ⟨*item*⟩ contains {, }, or # (assuming the usual TEX category codes apply).

23.5 Mapping over comma lists

The functions described in this section apply a specified function to each item of a comma list. All mappings are done at the current group level, *i.e.* any local assignments made by the ⟨*function*⟩ or ⟨*code*⟩ discussed below remain in effect after the loop.

When the comma list is given explicitly, as an n-type argument, spaces are trimmed around each item. If the result of trimming spaces is empty, the item is ignored. Otherwise, if the item is surrounded by braces, one set is removed, and the result is passed to the mapped function. Thus, if the comma list that is being mapped is {a␣,␣{{b}␣},␣,{},␣{c},} then the arguments passed to the mapped function are 'a', '{b}␣', an empty argument, and 'c'.

When the comma list is given as an N-type argument, spaces have already been trimmed on input, and items are simply stripped of one set of braces if any. This case is more efficient than using n-type comma lists.

\clist_map_function:NN ☆
\clist_map_function:cN ☆
\clist_map_function:nN ☆
\clist_map_function:eN ☆

Updated: 2012-06-29

\clist_map_function:NN ⟨clist var⟩ ⟨function⟩

Applies ⟨function⟩ to every ⟨item⟩ stored in the ⟨clist var⟩. The ⟨function⟩ receives one argument for each iteration. The ⟨items⟩ are returned from left to right. The function \clist_map_inline:Nn is in general more efficient than \clist_map_function:NN.

\clist_map_inline:Nn
\clist_map_inline:cn
\clist_map_inline:nn

Updated: 2012-06-29

\clist_map_inline:Nn ⟨clist var⟩ {⟨inline function⟩}

Applies ⟨inline function⟩ to every ⟨item⟩ stored within the ⟨clist var⟩. The ⟨inline function⟩ should consist of code which receives the ⟨item⟩ as #1. The ⟨items⟩ are returned from left to right.

\clist_map_variable:NNn
\clist_map_variable:cNn
\clist_map_variable:nNn

Updated: 2012-06-29

\clist_map_variable:NNn ⟨clist var⟩ ⟨variable⟩ {⟨code⟩}

Stores each ⟨item⟩ of the ⟨clist var⟩ in turn in the (token list) ⟨variable⟩ and applies the ⟨code⟩. The ⟨code⟩ will usually make use of the ⟨variable⟩, but this is not enforced. The assignments to the ⟨variable⟩ are local. Its value after the loop is the last ⟨item⟩ in the ⟨clist var⟩, or its original value if there were no ⟨item⟩. The ⟨items⟩ are returned from left to right.

\clist_map_tokens:Nn ☆
\clist_map_tokens:cn ☆
\clist_map_tokens:nn ☆

New: 2021-05-05

\clist_map_tokens:Nn ⟨clist var⟩ {⟨code⟩}
\clist_map_tokens:nn {⟨comma list⟩} {⟨code⟩}

Calls ⟨code⟩ {⟨item⟩} for every ⟨item⟩ stored in the ⟨clist var⟩. The ⟨code⟩ receives each ⟨item⟩ as a trailing brace group. If the ⟨code⟩ consists of a single function this is equivalent to \clist_map_function:nN.

\clist_map_break: ☆

Updated: 2012-06-29

\clist_map_break:

Used to terminate a \clist_map_... function before all entries in the ⟨comma list⟩ have been processed. This normally takes place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \clist_map_break: }
      {
        % Do something useful
      }
  }
```

Use outside of a \clist_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

\clist_map_break:n ☆ \clist_map_break:n {⟨*code*⟩}

Updated: 2012-06-29

Used to terminate a \clist_map_... function before all entries in the ⟨*comma list*⟩ have been processed, inserting the ⟨*code*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \clist_map_break:n { <code> } }
      {
        % Do something useful
      }
  }
```

Use outside of a \clist_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the ⟨*code*⟩ is inserted into the input stream. This depends on the design of the mapping function.

\clist_count:N ⋆
\clist_count:c ⋆
\clist_count:n ⋆
\clist_count:e ⋆

New: 2012-07-13

\clist_count:N ⟨*clist var*⟩

Leaves the number of items in the ⟨*clist var*⟩ in the input stream as an ⟨*integer denotation*⟩. The total number of items in a ⟨*clist var*⟩ includes those which are duplicates, *i.e.* every item in a ⟨*clist var*⟩ is counted.

23.6 Using the content of comma lists directly

`\clist_use:Nnnn` ★
`\clist_use:cnnn` ★

New: 2013-05-26

`\clist_use:Nnnn` ⟨*clist var*⟩ {⟨*separator between two*⟩} {⟨*separator between more than two*⟩} {⟨*separator between final two*⟩}

Places the contents of the ⟨*clist var*⟩ in the input stream, with the appropriate ⟨*separator*⟩ between the items. Namely, if the comma list has more than two items, the ⟨*separator between more than two*⟩ is placed between each pair of items except the last, for which the ⟨*separator between final two*⟩ is used. If the comma list has exactly two items, then they are placed in the input stream separated by the ⟨*separator between two*⟩. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nnnn \l_tmpa_clist { ~and~ } { ,~ } { ,~and~ }
```

inserts "`a, b, c, de, and f`" in the input stream. The first separator argument is not used in this case because the comma list has more than 2 items.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*items*⟩ do not expand further when appearing in an `e`-type or `x`-type argument expansion.

`\clist_use:Nn` ★
`\clist_use:cn` ★

New: 2013-05-26

`\clist_use:Nn` ⟨*clist var*⟩ {⟨*separator*⟩}

Places the contents of the ⟨*clist var*⟩ in the input stream, with the ⟨*separator*⟩ between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nn \l_tmpa_clist { ~and~ }
```

inserts "`a and b and c and de and f`" in the input stream.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*items*⟩ do not expand further when appearing in an `e`-type or `x`-type argument expansion.

\clist_use:nnnn ⋆
\clist_use:nn ⋆

New: 2021-05-10

\clist_use:nnnn {⟨*comma list*⟩} {⟨*separator between two*⟩}
 {⟨*separator between more than two*⟩} {⟨*separator between final two*⟩}
\clist_use:nn {⟨*comma list*⟩} {⟨*separator*⟩}

Places the contents of the ⟨*comma list*⟩ in the input stream, with the appropriate ⟨*separator*⟩ between the items. As for \clist_set:Nn, blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. The ⟨*separators*⟩ are then inserted in the same way as for \clist_use:Nnnn and \clist_use:Nn, respectively.

TEXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨*items*⟩ do not expand further when appearing in an e-type or x-type argument expansion.

23.7 Comma lists as stacks

Comma lists can be used as stacks, where data is pushed to and popped from the top of the comma list. (The left of a comma list is the top, for performance reasons.) The stack functions for comma lists are not intended to be mixed with the general ordered data functions detailed in the previous section: a comma list should either be used as an ordered data type or as a stack, but not in both ways.

\clist_get:NN
\clist_get:cN
\clist_get:NN*TF*
\clist_get:cN*TF*

New: 2012-05-14
Updated: 2019-02-16

\clist_get:NN ⟨*clist var*⟩ ⟨*tl var*⟩

Stores the left-most item from a ⟨*clist var*⟩ in the ⟨*tl var*⟩ without removing it from the ⟨*clist var*⟩. The ⟨*tl var*⟩ is assigned locally. In the non-branching version, if the ⟨*clist var*⟩ is empty the ⟨*tl var*⟩ is set to the marker value \q_no_value.

\clist_pop:NN
\clist_pop:cN

Updated: 2011-09-06

\clist_pop:NN ⟨*clist var*⟩ ⟨*tl var*⟩

Pops the left-most item from a ⟨*clist var*⟩ into the ⟨*tl var*⟩, *i.e.* removes the item from the comma list and stores it in the ⟨*tl var*⟩. Both of the variables are assigned locally.

\clist_gpop:NN
\clist_gpop:cN

\clist_gpop:NN ⟨*clist var*⟩ ⟨*tl var*⟩

Pops the left-most item from a ⟨*clist var*⟩ into the ⟨*tl var*⟩, *i.e.* removes the item from the comma list and stores it in the ⟨*tl var*⟩. The ⟨*clist var*⟩ is modified globally, while the assignment of the ⟨*tl var*⟩ is local.

\clist_pop:NN*TF*
\clist_pop:cN*TF*

New: 2012-05-14

\clist_pop:NNTF ⟨*clist var*⟩ ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*clist var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*clist var*⟩ is non-empty, pops the top item from the ⟨*clist var*⟩ in the ⟨*tl var*⟩, *i.e.* removes the item from the ⟨*clist var*⟩. Both the ⟨*clist var*⟩ and the ⟨*tl var*⟩ are assigned locally.

`\clist_gpop:NNTF`
`\clist_gpop:cNTF`

New: 2012-05-14

`\clist_gpop:NNTF` ⟨*clist var*⟩ ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*clist var*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*clist var*⟩ is non-empty, pops the top item from the ⟨*clist var*⟩ in the ⟨*tl var*⟩, *i.e.* removes the item from the ⟨*clist var*⟩. The ⟨*clist var*⟩ is modified globally, while the ⟨*tl var*⟩ is assigned locally.

`\clist_push:Nn`
`\clist_push:(NV|No|cn|cV|co)`
`\clist_gpush:Nn`
`\clist_gpush:(NV|No|cn|cV|co)`

`\clist_push:Nn` ⟨*clist var*⟩ {⟨*items*⟩}

Adds the {⟨*items*⟩} to the top of the ⟨*clist var*⟩. Spaces are removed from both sides of each item as for any `n`-type comma list.

23.8 Using a single item

`\clist_item:Nn` ⋆
`\clist_item:cn` ⋆
`\clist_item:nn` ⋆
`\clist_item:en` ⋆

New: 2014-07-17

`\clist_item:Nn` ⟨*clist var*⟩ {⟨*int expr*⟩}

Indexing items in the ⟨*clist var*⟩ from 1 at the top (left), this function evaluates the ⟨*int expr*⟩ and leaves the appropriate item from the comma list in the input stream. If the ⟨*int expr*⟩ is negative, indexing occurs from the bottom (right) of the comma list. When the ⟨*int expr*⟩ is larger than the number of items in the ⟨*clist var*⟩ (as calculated by `\clist_count:N`) then the function expands to nothing.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*item*⟩ does not expand further when appearing in an `e`-type or `x`-type argument expansion.

`\clist_rand_item:N` ⋆
`\clist_rand_item:c` ⋆
`\clist_rand_item:n` ⋆

New: 2016-12-06

`\clist_rand_item:N` ⟨*clist var*⟩
`\clist_rand_item:n` {⟨*comma list*⟩}

Selects a pseudo-random item of the ⟨*clist var*⟩/⟨*comma list*⟩. If the ⟨*comma list*⟩ has no item, the result is empty.

TeXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*item*⟩ does not expand further when appearing in an `e`-type or `x`-type argument expansion.

23.9 Viewing comma lists

`\clist_show:N`
`\clist_show:c`

Updated: 2021-04-29

`\clist_show:N` ⟨*clist var*⟩

Displays the entries in the ⟨*clist var*⟩ in the terminal.

\clist_show:n

Updated: 2013-08-03

\clist_show:n {⟨*tokens*⟩}

Displays the entries in the comma list in the terminal.

\clist_log:N
\clist_log:c

New: 2014-08-22
Updated: 2021-04-29

\clist_log:N ⟨*clist var*⟩

Writes the entries in the ⟨*clist var*⟩ in the log file. See also \clist_show:N which displays the result in the terminal.

\clist_log:n

New: 2014-08-22

\clist_log:n {⟨*tokens*⟩}

Writes the entries in the comma list in the log file. See also \clist_show:n which displays the result in the terminal.

23.10 Constant and scratch comma lists

\c_empty_clist

New: 2012-07-02

Constant that is always empty.

\l_tmpa_clist
\l_tmpb_clist

New: 2011-09-06

Scratch comma lists for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_clist
\g_tmpb_clist

New: 2011-09-06

Scratch comma lists for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Chapter 24

The l3token module
Token manipulation

This module deals with tokens. Now this is perhaps not the most precise description so let's try with a better description: When programming in TeX, it is often desirable to know just what a certain token is: is it a control sequence or something else. Similarly one often needs to know if a control sequence is expandable or not, a macro or a primitive, how many arguments it takes etc. Another thing of great importance (especially when it comes to document commands) is looking ahead in the token stream to see if a certain character is present and maybe even remove it or disregard other tokens while scanning. This module provides functions for both and as such has two primary function categories: `\token_` for anything that deals with tokens and `\peek_` for looking ahead in the token stream.

Most functions we describe here can be used on control sequences, as those are tokens as well.

It is important to distinguish two aspects of a token: its "shape" (for lack of a better word), which affects the matching of delimited arguments and the comparison of token lists containing this token, and its "meaning", which affects whether the token expands or what operation it performs. One can have tokens of different shapes with the same meaning, but not the converse.

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three names for the same internal operation of TeX, namely the primitive testing the next two characters for equality of their character code. They have the same meaning hence behave identically in many situations. However, TeX distinguishes them when searching for a delimited argument. Namely, the example function `\show_until_if:w` defined below takes everything until `\if:w` as an argument, despite the presence of other copies of `\if:w` under different names.

```
\cs_new:Npn \show_until_if:w #1 \if:w { \tl_show:n {#1} }
\show_until_if:w \tex_if:D \if_charcode:w \if:w
```

A list of all possible shapes and a list of all possible meanings are given in section 24.7.

24.1 Creating character tokens

\char_set_active_eq:NN
\char_set_active_eq:Nc
\char_gset_active_eq:NN
\char_gset_active_eq:Nc

Updated: 2015-11-12

\char_set_active_eq:NN ⟨*char*⟩ ⟨*function*⟩

Sets the behaviour of the ⟨*char*⟩ in situations where it is active (category code 13) to be equivalent to that of the definition of the ⟨*function*⟩ at the time \char_set_active_eq:NN is used. The category code of the ⟨*char*⟩ is *unchanged* by this process. The ⟨*function*⟩ may itself be an active character.

\char_set_active_eq:nN
\char_set_active_eq:nc
\char_gset_active_eq:nN
\char_gset_active_eq:nc

New: 2015-11-12

\char_set_active_eq:nN {⟨*integer expression*⟩} ⟨*function*⟩

Sets the behaviour of the ⟨*char*⟩ which has character code as given by the ⟨*integer expression*⟩ in situations where it is active (category code 13) to be equivalent to that of the ⟨*function*⟩ at the time \char_set_active_eq:nN is used. The category code of the ⟨*char*⟩ is *unchanged* by this process. The ⟨*function*⟩ may itself be an active character.

\char_generate:nn ⋆

New: 2015-09-09
Updated: 2019-01-16

\char_generate:nn {⟨*charcode*⟩} {⟨*catcode*⟩}

Generates a character token of the given ⟨*charcode*⟩ and ⟨*catcode*⟩ (both of which may be integer expressions). The ⟨*catcode*⟩ may be one of

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 10 (space)
- 11 (letter)
- 12 (other)
- 13 (active)

and other values raise an error. The ⟨*charcode*⟩ may be any one valid for the engine in use, except that for ⟨*catcode*⟩ 10, ⟨*charcode*⟩ 0 is not allowed. Active characters cannot be generated in older versions of XeTeX. Another way to build token lists with unusual category codes is \regex_replace:nnN {.*} {⟨*replacement*⟩} ⟨*tl var*⟩.

TeXhackers note: Exactly two expansions are needed to produce the character.

\c_catcode_active_space_tl

New: 2017-08-07

Token list containing one character with category code 13, (“active”), and character code 32 (space).

`\c_catcode_other_space_tl`

New: 2011-09-05

Token list containing one character with category code 12, ("other"), and character code 32 (space).

24.2 Manipulating and interrogating character tokens

```
\char_set_catcode_escape:N
\char_set_catcode_group_begin:N
\char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N
\char_set_catcode_alignment:N
\char_set_catcode_end_line:N
\char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
\char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N
```

Updated: 2015-11-11

`\char_set_catcode_letter:N` ⟨*character*⟩

Sets the category code of the ⟨*character*⟩ to that indicated in the function name. Depending on the current category code of the ⟨*token*⟩ the escape token may also be needed:

```
\char_set_catcode_other:N \%
```

The assignment is local.

```
\char_set_catcode_escape:n
\char_set_catcode_group_begin:n
\char_set_catcode_group_end:n
\char_set_catcode_math_toggle:n
\char_set_catcode_alignment:n
\char_set_catcode_end_line:n
\char_set_catcode_parameter:n
\char_set_catcode_math_superscript:n
\char_set_catcode_math_subscript:n
\char_set_catcode_ignore:n
\char_set_catcode_space:n
\char_set_catcode_letter:n
\char_set_catcode_other:n
\char_set_catcode_active:n
\char_set_catcode_comment:n
\char_set_catcode_invalid:n
```

Updated: 2015-11-11

`\char_set_catcode_letter:n {⟨integer expression⟩}`

Sets the category code of the ⟨*character*⟩ which has character code as given by the ⟨*integer expression*⟩. This version can be used to set up characters which cannot otherwise be given (*cf.* the N-type variants). The assignment is local.

`\char_set_catcode:nn`

Updated: 2015-11-11

`\char_set_catcode:nn {⟨int expr₁⟩} {⟨int expr₂⟩}`

These functions set the category code of the ⟨*character*⟩ which has character code as given by the ⟨*integer expression*⟩. The first ⟨*integer expression*⟩ is the character code and the second is the category code to apply. The setting applies within the current TeX group. In general, the symbolic functions `\char_set_catcode_⟨type⟩` should be preferred, but there are cases where these lower-level functions may be useful.

`\char_value_catcode:n` ⋆

`\char_value_catcode:n {⟨integer expression⟩}`

Expands to the current category code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_catcode:n`

`\char_show_value_catcode:n {⟨integer expression⟩}`

Displays the current category code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_lccode:nn`

Updated: 2015-08-06

`\char_set_lccode:nn {⟨int expr₁⟩} {⟨int expr₂⟩}`

Sets up the behaviour of the ⟨*character*⟩ when found inside `\text_lowercase:n`, such that ⟨*character*$_1$⟩ will be converted into ⟨*character*$_2$⟩. The two ⟨*characters*⟩ may be specified using an ⟨*integer expression*⟩ for the character code concerned. This may include the TeX `‘⟨character⟩` method for converting a single character into its character code:

```
\char_set_lccode:nn { `\A } { `\a } % Standard behaviour
\char_set_lccode:nn { `\A } { `\A + 32 }
\char_set_lccode:nn { 50 } { 60 }
```

The setting applies within the current TeX group.

`\char_value_lccode:n` ⋆ `\char_value_lccode:n {⟨integer expression⟩}`

Expands to the current lower case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_lccode:n` `\char_show_value_lccode:n {⟨integer expression⟩}`

Displays the current lower case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_uccode:nn` `\char_set_uccode:nn {⟨int expr1⟩} {⟨int expr2⟩}`

Updated: 2015-08-06

Sets up the behaviour of the ⟨*character*⟩ when found inside `\text_uppercase:n`, such that ⟨*character*$_1$⟩ will be converted into ⟨*character*$_2$⟩. The two ⟨*characters*⟩ may be specified using an ⟨*integer expression*⟩ for the character code concerned. This may include the TeX `‘⟨character⟩` method for converting a single character into its character code:

```
\char_set_uccode:nn { ‘\a } { ‘\A } % Standard behaviour
\char_set_uccode:nn { ‘\A } { ‘\A - 32 }
\char_set_uccode:nn { 60 } { 50 }
```

The setting applies within the current TeX group.

`\char_value_uccode:n` ⋆ `\char_value_uccode:n {⟨integer expression⟩}`

Expands to the current upper case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_uccode:n` `\char_show_value_uccode:n {⟨integer expression⟩}`

Displays the current upper case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_mathcode:nn` `\char_set_mathcode:nn {⟨int expr1⟩} {⟨int expr2⟩}`

Updated: 2015-08-06

This function sets up the math code of ⟨*character*⟩. The ⟨*character*⟩ is specified as an ⟨*integer expression*⟩ which will be used as the character code of the relevant character. The setting applies within the current TeX group.

`\char_value_mathcode:n` ⋆ `\char_value_mathcode:n {⟨integer expression⟩}`

Expands to the current math code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_mathcode:n` `\char_show_value_mathcode:n {⟨integer expression⟩}`

Displays the current math code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_sfcode:nn` `\char_set_sfcode:nn {⟨int expr1⟩} {⟨int expr2⟩}`

Updated: 2015-08-06

This function sets up the space factor for the ⟨*character*⟩. The ⟨*character*⟩ is specified as an ⟨*integer expression*⟩ which will be used as the character code of the relevant character. The setting applies within the current TeX group.

\char_value_sfcode:n ⋆ \char_value_sfcode:n {⟨*integer expression*⟩}

Expands to the current space factor for the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

\char_show_value_sfcode:n \char_show_value_sfcode:n {⟨*integer expression*⟩}

Displays the current space factor for the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

\l_char_active_seq

New: 2012-01-23
Updated: 2015-11-11

Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category ⟨*active*⟩ (catcode 13). Each entry in the sequence consists of a single escaped token, for example \~. Active tokens should be added to the sequence when they are defined for general document use.

\l_char_special_seq

New: 2012-01-23
Updated: 2015-11-11

Used to track which tokens will require special handling when working with verbatim-like material at the document level as they are not of categories ⟨*letter*⟩ (catcode 11) or ⟨*other*⟩ (catcode 12). Each entry in the sequence consists of a single escaped token, for example \\ for the backslash or \{ for an opening brace. Escaped tokens should be added to the sequence when they are defined for general document use.

24.3 Generic tokens

\c_group_begin_token
\c_group_end_token
\c_math_toggle_token
\c_alignment_token
\c_parameter_token
\c_math_superscript_token
\c_math_subscript_token
\c_space_token

These are implicit tokens which have the category code described by their name. They are used internally for test purposes but are also available to the programmer for other uses.

TEXhackers note: The tokens \c_group_begin_token, \c_group_end_token, and \c_-space_token are expl3 counterparts of LaTeX 2_ε's \bgroup, \egroup, and \@sptoken.

\c_catcode_letter_token
\c_catcode_other_token

These are implicit tokens which have the category code described by their name. They are used internally for test purposes and should not be used other than for category code tests.

\c_catcode_active_tl A token list containing an active token. This is used internally for test purposes and should not be used other than in appropriately-constructed category code tests.

24.4 Converting tokens

\token_to_meaning:N ⋆
\token_to_meaning:c ⋆

\token_to_meaning:N ⟨*token*⟩

Inserts the current meaning of the ⟨*token*⟩ into the input stream as a series of characters of category code 12 (other). This is the primitive TeX description of the ⟨*token*⟩, thus for example both functions defined by `\cs_set_nopar:Npn` and token list variables defined using `\tl_new:N` are described as `macros`.

TeXhackers note: This is the TeX primitive `\meaning`. The ⟨*token*⟩ can thus be an explicit space token or an explicit begin-group or end-group character token (`{` or `}` when normal TeX category codes apply) even though these are not valid `N`-type arguments.

\token_to_str:N ⋆
\token_to_str:c ⋆

\token_to_str:N ⟨*token*⟩

Converts the given ⟨*token*⟩ into a series of characters with category code 12 (other). If the ⟨*token*⟩ is a control sequence, this will start with the current escape character with category code 12 (the escape character is part of the ⟨*token*⟩). This function requires only a single expansion.

TeXhackers note: `\token_to_str:N` is the TeX primitive `\string`. The ⟨*token*⟩ can thus be an explicit space tokens or an explicit begin-group or end-group character token (`{` or `}` when normal TeX category codes apply) even though these are not valid `N`-type arguments.

\token_to_catcode:N ⋆

New: 2023-10-15

\token_to_catcode:N ⟨*token*⟩

Converts the given ⟨*token*⟩ into a number describing its category code. If ⟨*token*⟩ is a control sequence this expands to 16. This can't detect the categories 0 (escape character), 5 (end of line), 9 (ignored character), 14 (comment character), or 15 (invalid character). Control sequences or active characters let to a token of one of the detectable category codes will yield that category.

24.5 Token conditionals

\token_if_group_begin_p:N ⋆
\token_if_group_begin:N*TF* ⋆

\token_if_group_begin_p:N ⟨*token*⟩
\token_if_group_begin:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of a begin group token (`{` when normal TeX category codes are in force). Note that an explicit begin group token cannot be tested in this way, as it is not a valid `N`-type argument.

\token_if_group_end_p:N ⋆
\token_if_group_end:N*TF* ⋆

\token_if_group_end_p:N ⟨*token*⟩
\token_if_group_end:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of an end group token (`}` when normal TeX category codes are in force). Note that an explicit end group token cannot be tested in this way, as it is not a valid `N`-type argument.

`\token_if_math_toggle_p:N` ⋆
`\token_if_math_toggle:NTF` ⋆

`\token_if_math_toggle_p:N` ⟨*token*⟩
`\token_if_math_toggle:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of a math shift token (`$` when normal TeX category codes are in force).

`\token_if_alignment_p:N` ⋆
`\token_if_alignment:NTF` ⋆

`\token_if_alignment_p:N` ⟨*token*⟩
`\token_if_alignment:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of an alignment token (`&` when normal TeX category codes are in force).

`\token_if_parameter_p:N` ⋆
`\token_if_parameter:NTF` ⋆

`\token_if_parameter_p:N` ⟨*token*⟩
`\token_if_parameter:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of a macro parameter token (`#` when normal TeX category codes are in force).

`\token_if_math_superscript_p:N` ⋆
`\token_if_math_superscript:NTF` ⋆

`\token_if_math_superscript_p:N` ⟨*token*⟩
`\token_if_math_superscript:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of a superscript token (`^` when normal TeX category codes are in force).

`\token_if_math_subscript_p:N` ⋆
`\token_if_math_subscript:NTF` ⋆

`\token_if_math_subscript_p:N` ⟨*token*⟩
`\token_if_math_subscript:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of a subscript token (`_` when normal TeX category codes are in force).

`\token_if_space_p:N` ⋆
`\token_if_space:NTF` ⋆

`\token_if_space_p:N` ⟨*token*⟩
`\token_if_space:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of a space token. Note that an explicit space token with character code 32 cannot be tested in this way, as it is not a valid `N`-type argument.

`\token_if_letter_p:N` ⋆
`\token_if_letter:NTF` ⋆

`\token_if_letter_p:N` ⟨*token*⟩
`\token_if_letter:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of a letter token.

`\token_if_other_p:N` ⋆
`\token_if_other:NTF` ⋆

`\token_if_other_p:N` ⟨*token*⟩
`\token_if_other:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of an "other" token.

`\token_if_active_p:N` ⋆
`\token_if_active:NTF` ⋆

`\token_if_active_p:N` ⟨*token*⟩
`\token_if_active:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of an active character.

`\token_if_eq_catcode_p:NN` ⋆
`\token_if_eq_catcode:NNTF` ⋆

`\token_if_eq_catcode_p:NN` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩
`\token_if_eq_catcode:NNTF` ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the two ⟨*tokens*⟩ have the same category code.

\token_if_eq_charcode_p:NN ⋆
\token_if_eq_charcode:NNTF ⋆

\token_if_eq_charcode_p:NN ⟨token$_1$⟩ ⟨token$_2$⟩
\token_if_eq_charcode:NNTF ⟨token$_1$⟩ ⟨token$_2$⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the two ⟨tokens⟩ have the same character code.

\token_if_eq_meaning_p:NN ⋆
\token_if_eq_meaning:NNTF ⋆

\token_if_eq_meaning_p:NN ⟨token$_1$⟩ ⟨token$_2$⟩
\token_if_eq_meaning:NNTF ⟨token$_1$⟩ ⟨token$_2$⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the two ⟨tokens⟩ have the same meaning when expanded.

\token_if_macro_p:N ⋆
\token_if_macro:NTF ⋆

Updated: 2011-05-23

\token_if_macro_p:N ⟨token⟩
\token_if_macro:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨token⟩ is a TEX macro.

\token_if_cs_p:N ⋆
\token_if_cs:NTF ⋆

\token_if_cs_p:N ⟨token⟩
\token_if_cs:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨token⟩ is a control sequence.

\token_if_expandable_p:N ⋆
\token_if_expandable:NTF ⋆

\token_if_expandable_p:N ⟨token⟩
\token_if_expandable:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨token⟩ is expandable. This test returns ⟨false⟩ for an undefined token.

\token_if_long_macro_p:N ⋆
\token_if_long_macro:NTF ⋆

Updated: 2012-01-20

\token_if_long_macro_p:N ⟨token⟩
\token_if_long_macro:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨token⟩ is a long macro with no other prefix; to test for a macro that is both long and protected, use \token_if_protected_long_macro:N(TF).

\token_if_protected_macro_p:N ⋆
\token_if_protected_macro:NTF ⋆

Updated: 2012-01-20

\token_if_protected_macro_p:N ⟨token⟩
\token_if_protected_macro:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨token⟩ is a protected macro with no other prefix; to test for a macro that is both protected and long, use \token_if_protected_long_macro:N(TF).

\token_if_protected_long_macro_p:N ⋆
\token_if_protected_long_macro:NTF ⋆

Updated: 2012-01-20

\token_if_protected_long_macro_p:N ⟨token⟩
\token_if_protected_long_macro:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨token⟩ is a protected long macro.

\token_if_chardef_p:N ⋆
\token_if_chardef:NTF ⋆

Updated: 2012-01-20

\token_if_chardef_p:N ⟨token⟩
\token_if_chardef:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨token⟩ is defined to be a chardef.

TEXhackers note: Booleans, boxes and small integer constants are implemented as \chardefs.

\token_if_mathchardef_p:N ⋆ \token_if_mathchardef_p:N ⟨*token*⟩
\token_if_mathchardef:N*TF* ⋆ \token_if_mathchardef:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Updated: 2012-01-20

Tests if the ⟨*token*⟩ is defined to be a mathchardef.

\token_if_font_selection_p:N ⋆ \token_if_font_selection_p:N ⟨*token*⟩
\token_if_font_selection:N*TF* ⋆ \token_if_font_selection:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

New: 2020-10-27

Tests if the ⟨*token*⟩ is defined to be a font selection command.

\token_if_dim_register_p:N ⋆ \token_if_dim_register_p:N ⟨*token*⟩
\token_if_dim_register:N*TF* ⋆ \token_if_dim_register:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Updated: 2012-01-20

Tests if the ⟨*token*⟩ is defined to be a dimension register.

\token_if_int_register_p:N ⋆ \token_if_int_register_p:N ⟨*token*⟩
\token_if_int_register:N*TF* ⋆ \token_if_int_register:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Updated: 2012-01-20

Tests if the ⟨*token*⟩ is defined to be a integer register.

TEXhackers note: Constant integers may be implemented as integer registers, \chardefs, or \mathchardefs depending on their value.

\token_if_muskip_register_p:N ⋆ \token_if_muskip_register_p:N ⟨*token*⟩
\token_if_muskip_register:N*TF* ⋆ \token_if_muskip_register:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

New: 2012-02-15

Tests if the ⟨*token*⟩ is defined to be a muskip register.

\token_if_skip_register_p:N ⋆ \token_if_skip_register_p:N ⟨*token*⟩
\token_if_skip_register:N*TF* ⋆ \token_if_skip_register:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Updated: 2012-01-20

Tests if the ⟨*token*⟩ is defined to be a skip register.

\token_if_toks_register_p:N ⋆ \token_if_toks_register_p:N ⟨*token*⟩
\token_if_toks_register:N*TF* ⋆ \token_if_toks_register:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Updated: 2012-01-20

Tests if the ⟨*token*⟩ is defined to be a toks register (not used by LaTeX3).

\token_if_primitive_p:N ⋆ \token_if_primitive_p:N ⟨*token*⟩
\token_if_primitive:N*TF* ⋆ \token_if_primitive:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Updated: 2020-09-11

Tests if the ⟨*token*⟩ is an engine primitive. In LuaTeX this includes primitive-like commands defined using token.set_lua.

\token_case_catcode:Nn ⋆
\token_case_catcode:Nn*TF* ⋆
\token_case_charcode:Nn ⋆
\token_case_charcode:Nn*TF* ⋆
\token_case_meaning:Nn ⋆
\token_case_meaning:Nn*TF* ⋆

New: 2020-12-03

```
\token_case_meaning:NnTF ⟨test token⟩
  {
    ⟨token case1⟩ {⟨code case1⟩}
    ⟨token case2⟩ {⟨code case2⟩}
    ...
    ⟨token casen⟩ {⟨code casen⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function compares the ⟨*test token*⟩ in turn with each of the ⟨*token case*⟩s. If the two are equal (as described for \token_if_eq_catcode:NNTF, \token_if_eq_-charcode:NNTF and \token_if_eq_meaning:NNTF, respectively) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The functions \token_case_catcode:Nn, \token_case_charcode:Nn, and \token_case_meaning:Nn, which do nothing if there is no match, are also available.

24.6 Peeking ahead at the next token

There is often a need to look ahead at the next token in the input stream while leaving it in place. This is handled using the "peek" functions. The generic \peek_after:Nw is provided along with a family of predefined tests for common cases. Peeking ahead does *not* skip spaces: rather, \peek_remove_spaces:n. should be used. In addition, using \peek_analysis_map_inline:n, one can map through the following tokens in the input stream and repeatedly perform some tests.

\peek_after:Nw

\peek_after:Nw ⟨*function*⟩ ⟨*token*⟩

Locally sets the test variable \l_peek_token equal to ⟨*token*⟩ (as an implicit token, *not* as a token list), and then expands the ⟨*function*⟩. The ⟨*token*⟩ remains in the input stream as the next item after the ⟨*function*⟩. The ⟨*token*⟩ here may be ␣, { or } (assuming normal TeX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

\peek_gafter:Nw

\peek_gafter:Nw ⟨*function*⟩ ⟨*token*⟩

Globally sets the test variable \g_peek_token equal to ⟨*token*⟩ (as an implicit token, *not* as a token list), and then expands the ⟨*function*⟩. The ⟨*token*⟩ remains in the input stream as the next item after the ⟨*function*⟩. The ⟨*token*⟩ here may be ␣, { or } (assuming normal TeX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

\l_peek_token

Token set by \peek_after:Nw and available for testing as described above.

\g_peek_token

Token set by \peek_gafter:Nw and available for testing as described above.

`\peek_catcode:NTF` Updated: 2012-12-20

`\peek_catcode:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ is left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

`\peek_catcode_remove:NTF` Updated: 2012-12-20

`\peek_catcode_remove:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ is removed from the input stream if the test is true. The function then places either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

`\peek_charcode:NTF` Updated: 2012-12-20

`\peek_charcode:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ is left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

`\peek_charcode_remove:NTF` Updated: 2012-12-20

`\peek_charcode_remove:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ is removed from the input stream if the test is true. The function then places either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

`\peek_meaning:NTF` Updated: 2011-07-02

`\peek_meaning:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ is left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

`\peek_meaning_remove:NTF` Updated: 2011-07-02

`\peek_meaning_remove:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ is removed from the input stream if the test is true. The function then places either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

`\peek_remove_spaces:n` New: 2018-10-01

`\peek_remove_spaces:n` {⟨*code*⟩}

Peeks ahead and detect if the following token is a space (category code 10 and character code 32). If so, removes the token and checks the next token. Once a non-space token is found, the ⟨*code*⟩ will be inserted into the input stream. Typically this will contain a `peek` operation, but this is not required.

\peek_remove_filler:n `\peek_remove_filler:n {⟨code⟩}`

New: 2022-01-10

Peeks ahead and detect if the following token is a space (category code 10) or has meaning equal to `\scan_stop:`. If so, removes the token and checks the next token. If neither of these cases apply, expands the next token using `f`-type expansion, then checks the resulting leading token in the same way. If after expansion the next token is neither of the two test cases, the ⟨*code*⟩ will be inserted into the input stream. Typically this will contain a `peek` operation, but this is not required.

TEXhackers note: This is essentially a macro-based implementation of how TEX handles the search for a left brace after for example `\everypar`, except that any non-expandable token cleanly ends the ⟨*filler*⟩ (i.e. it does not lead to a TEX error).

In contrast to TEX's filler removal, a construct `\exp_not:N \foo` will be treated in the same way as `\foo`.

\peek_N_type:*TF* `\peek_N_type:TF {⟨true code⟩} {⟨false code⟩}`

Updated: 2012-12-20

Tests if the next ⟨*token*⟩ in the input stream can be safely grabbed as an `N`-type argument. The test is ⟨*false*⟩ if the next ⟨*token*⟩ is either an explicit or implicit begin-group or end-group token (with any character code), or an explicit or implicit space character (with character code 32 and category code 10), or an outer token (never used in LATEX3) and ⟨*true*⟩ in all other cases. Note that a ⟨*true*⟩ result ensures that the next ⟨*token*⟩ is a valid `N`-type argument. However, if the next ⟨*token*⟩ is for instance `\c_space_token`, the test takes the ⟨*false*⟩ branch, even though the next ⟨*token*⟩ is in fact a valid `N`-type argument. The ⟨*token*⟩ is left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

`\peek_analysis_map_inline:n` `\peek_analysis_map_inline:n {⟨inline function⟩}`

New: 2020-12-03
Updated: 2024-02-07

Repeatedly removes one ⟨*token*⟩ from the input stream and applies the ⟨*inline function*⟩ to it, until `\peek_analysis_map_break:` is called. The ⟨*inline function*⟩ receives three arguments for each ⟨*token*⟩ in the input stream:

- ⟨*tokens*⟩, which both `o`-expand and `e`/`x`-expand to the ⟨*token*⟩. The detailed form of ⟨*tokens*⟩ may change in later releases.
- ⟨*char code*⟩, a decimal representation of the character code of the ⟨*token*⟩, -1 if it is a control sequence.
- ⟨*catcode*⟩, a capital hexadecimal digit which denotes the category code of the ⟨*token*⟩ (0: control sequence, 1: begin-group, 2: end-group, 3: math shift, 4: alignment tab, 6: parameter, 7: superscript, 8: subscript, A: space, B: letter, C: other, D: active). This can be converted to an integer by writing `"`⟨*catcode*⟩.

These arguments are the same as for `\tl_analysis_map_inline:nn` defined in l3tl-analysis. The ⟨*char code*⟩ and ⟨*catcode*⟩ do not take the meaning of a control sequence or active character into account: for instance, upon encountering the token `\c_group_begin_token` in the input stream, `\peek_analysis_map_inline:n` calls the ⟨*inline function*⟩ with `#1` being `\exp_not:n { \c_group_begin_token }` (with the current implementation), `#2` being -1, and `#3` being 0, as for any other control sequence. In contrast, upon encountering an explicit begin-group token `{`, the ⟨*inline function*⟩ is called with arguments `\exp_after:wN { \if_false: } \fi:`, 123 and 1.

The mapping is done at the current group level, *i.e.* any local assignments made by the ⟨*inline function*⟩ remain in effect after the loop. Within the code, `\l_peek_token` is set equal (as a token, not a token list) to the token under consideration.

Peek functions cannot be used within this mapping function (nor other mapping functions) since the input stream contains trailing material necessary for the functioning of the loop.

TeXhackers note: In case the input stream has not yet been tokenized (converted from characters to tokens), characters are tokenized one by one as needed by `\peek_analysis_map_inline:n` using the current category code regime.

`\peek_analysis_map_break:`
`\peek_analysis_map_break:n`

```
\peek_analysis_map_inline:n
{ ...  \peek_analysis_map_break:n {⟨code⟩} }
```

New: 2020-12-03

Stops the `\peek_analysis_map_inline:n` loop from seeking more tokens, and inserts ⟨*code*⟩ in the input stream (empty for `\peek_analysis_map_break:`).

\peek_regex:n*TF*
\peek_regex:N*TF*

New: 2020-12-03

\peek_regex:nTF {⟨*regex*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*tokens*⟩ that follow in the input stream match the ⟨*regular expression*⟩. Any ⟨*tokens*⟩ that have been read are left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test). See l3regex for documentation of the syntax of regular expressions. The ⟨*regular expression*⟩ is implicitly anchored at the start, so for instance `\peek_regex:nTF { a }` is essentially equivalent to `\peek_charcode:NTF a`.

TEXhackers note: Implicit character tokens are correctly considered by `\peek_regex:nTF` as control sequences, while functions that inspect individual tokens (for instance `\peek_charcode:NTF`) only take into account their meaning.

The `\peek_regex:nTF` function only inspects as few tokens as necessary to determine whether the regular expression matches. For instance `\peek_regex:nTF { abc | [a-z] } { } { } abc` will only inspect the first token `a` even though the first branch `abc` of the alternative is preferred in functions such as `\peek_regex_remove_once:nTF`. This may have an effect on tokenization if the input stream has not yet been tokenized and category codes are changed.

\peek_regex_remove_once:n*TF*
\peek_regex_remove_once:N*TF*

New: 2020-12-03

\peek_regex_remove_once:nTF {⟨*regex*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*tokens*⟩ that follow in the input stream match the ⟨*regex*⟩. If the test is true, the ⟨*tokens*⟩ are removed from the input stream and the ⟨*true code*⟩ is inserted, while if the test is false, the ⟨*false code*⟩ is inserted followed by the ⟨*tokens*⟩ that were originally in the input stream. See l3regex for documentation of the syntax of regular expressions. The ⟨*regular expression*⟩ is implicitly anchored at the start, so for instance `\peek_regex_remove_once:nTF { a }` is essentially equivalent to `\peek_charcode_remove:NTF a`.

TEXhackers note: Implicit character tokens are correctly considered by `\peek_regex_remove_once:nTF` as control sequences, while functions that inspect individual tokens (for instance `\peek_charcode:NTF`) only take into account their meaning.

`\peek_regex_replace_once:nn`
`\peek_regex_replace_once:nnTF`
`\peek_regex_replace_once:Nn`
`\peek_regex_replace_once:NnTF`

New: 2020-12-03

`\peek_regex_replace_once:nnTF {⟨regex⟩} {⟨replacement⟩} {⟨true code⟩} {⟨false code⟩}`

If the ⟨*tokens*⟩ that follow in the input stream match the ⟨*regex*⟩, replaces them according to the ⟨*replacement*⟩ as for `\regex_replace_once:nnN`, and leaves the result in the input stream, after the ⟨*true code*⟩. Otherwise, leaves ⟨*false code*⟩ followed by the ⟨*tokens*⟩ that were originally in the input stream, with no modifications. See l3regex for documentation of the syntax of regular expressions and of the ⟨*replacement*⟩: for instance `\0` in the ⟨*replacement*⟩ is replaced by the tokens that were matched in the input stream. The ⟨*regular expression*⟩ is implicitly anchored at the start. In contrast to `\regex_replace_once:nnN`, no error arises if the ⟨*replacement*⟩ leads to an unbalanced token list: the tokens are inserted into the input stream without issue.

TEXhackers note: Implicit character tokens are correctly considered by `\peek_regex_replace_once:nnTF` as control sequences, while functions that inspect individual tokens (for instance `\peek_charcode:NTF`) only take into account their meaning.

24.7 Description of all possible tokens

Let us end by reviewing every case that a given token can fall into. This section is quite technical and some details are only meant for completeness. We distinguish the meaning of the token, which controls the expansion of the token and its effect on TEX's state, and its shape, which is used when comparing token lists such as for delimited arguments. Two tokens of the same shape must have the same meaning, but the converse does not hold.

A token has one of the following shapes.

- A control sequence, characterized by the sequence of characters that constitute its name: for instance, `\use:n` is a five-letter control sequence.
- An active character token, characterized by its character code (between 0 and 1114111 for LuaTEX and XeTEX and less for other engines) and category code 13.
- A character token, characterized by its character code and category code (one of 1, 2, 3, 4, 6, 7, 8, 10, 11 or 12 whose meaning is described below).

There are also a few internal tokens. The following list may be incomplete in some engines.

- Expanding `\the\font` results in a token that looks identical to the command that was used to select the current font (such as `\tenrm`) but it differs from it in shape.
- A "frozen" `\relax`, which differs from the primitive in shape (but has the same meaning), is inserted when the closing `\fi` of a conditional is encountered before the conditional is evaluated.
- Expanding `\noexpand` ⟨*token*⟩ (when the ⟨*token*⟩ is expandable) results in an internal token, displayed (temporarily) as `\notexpanded:` ⟨*token*⟩, whose shape coincides with the ⟨*token*⟩ and whose meaning differs from `\relax`.

- An `\outer endtemplate:` can be encountered when peeking ahead at the next token; this expands to another internal token, `end of alignment template`.
- Tricky programming might access a frozen `\endwrite`.
- Some frozen tokens can only be accessed in interactive sessions: `\cr`, `\right`, `\endgroup`, `\fi`, `\inaccessible`.
- In LuaTeX, there is also the strange case of "bytes" `^^^^^^1100`xy where x, y are any two lowercase hexadecimal digits, so that the hexadecimal number ranges from "11 0000 = 1 114 112 to "110 0ff = 1 114 367. These are used to output individual bytes to files, rather than UTF-8. For the purposes of token comparisons they behave like non-expandable primitive control sequences (*not characters*) whose `\meaning` is `the␣character␣` followed by the given byte. If this byte is in the range `80`–`ff` this gives an "invalid utf-8 sequence" error: applying `\token_to_str:N` or `\token_to_meaning:N` to these tokens is unsafe. Unfortunately, they don't seem to be detectable safely by any means except perhaps Lua code.

The meaning of a (non-active) character token is fixed by its category code (and character code) and cannot be changed. We call these tokens *explicit* character tokens. Category codes that a character token can have are listed below by giving a sample output of the TeX primitive `\meaning`, together with their LaTeX3 names and most common example:

1 begin-group character (`group_begin`, often `{`),

2 end-group character (`group_end`, often `}`),

3 math shift character (`math_toggle`, often `$`),

4 alignment tab character (`alignment`, often `&`),

6 macro parameter character (`parameter`, often `#`),

7 superscript character (`math_superscript`, often `^`),

8 subscript character (`math_subscript`, often `_`),

10 blank space (`space`, often character code 32),

11 the letter (`letter`, such as `A`),

12 the character (`other`, such as `0`).

Category code 13 (`active`) is discussed below. Input characters can also have several other category codes which do not lead to character tokens for later processing: 0 (`escape`), 5 (`end_line`), 9 (`ignore`), 14 (`comment`), and 15 (`invalid`).

The meaning of a control sequence or active character can be identical to that of any character token listed above (with any character code), and we call such tokens *implicit* character tokens. The meaning is otherwise in the following list:

- a macro, used in LaTeX3 for most functions and some variables (`tl`, `fp`, `seq`, ...),
- a primitive such as `\def` or `\topmark`, used in LaTeX3 for some functions,
- a register such as `\count123`, used in LaTeX3 for the implementation of some variables (`int`, `dim`, ...),

- a constant integer such as `\char"56` or `\mathchar"121`,
- a font selection command,
- undefined.

Macros can be `\protected` or not, `\long` or not (the opposite of what LaTeX3 calls `nopar`), and `\outer` or not (unused in LaTeX3). Their `\meaning` takes the form

⟨*prefix*⟩ `macro:`⟨*argument*⟩`->`⟨*replacement*⟩

where ⟨*prefix*⟩ is among `\protected\long\outer`, ⟨*argument*⟩ describes parameters that the macro expects, such as `#1#2#3`, and ⟨*replacement*⟩ describes how the parameters are manipulated, such as `\int_eval:n{#2+#1*#3}`.

Now is perhaps a good time to mention some subtleties relating to tokens with category code 10 (space). Any input character with this category code (normally, space and tab characters) becomes a normal space, with character code 32 and category code 10.

When a macro takes an undelimited argument, explicit space characters (with character code 32 and category code 10) are ignored. If the following token is an explicit character token with category code 1 (begin-group) and an arbitrary character code, then TeX scans ahead to obtain an equal number of explicit character tokens with category code 1 (begin-group) and 2 (end-group), and the resulting list of tokens (with outer braces removed) becomes the argument. Otherwise, a single token is taken as the argument for the macro: we call such single tokens "N-type", as they are suitable to be used as an argument for a function with the signature `:N`.

When a macro takes a delimited argument TeX scans ahead until finding the delimiter (outside any pairs of begin-group/end-group explicit characters), and the resulting list of tokens (with outer braces removed) becomes the argument. Note that explicit space characters at the start of the argument are *not* ignored in this case (and they prevent brace-stripping).

Chapter 25

The l3prop module Property lists

expl3 implements a "property list" data type, which contain an unordered list of entries each of which consists of a ⟨*key*⟩ (string) and an associated ⟨*value*⟩ (token list). The ⟨*key*⟩ and ⟨*value*⟩ may both be given as any balanced text, and the ⟨*key*⟩ is processed using `\tl_to_str:n`, meaning that category codes are ignored. Entries can be manipulated individually, as well as collectively by applying a function to every key–value pair within the list.

Each entry in a property list must have a unique ⟨*key*⟩: if an entry is added to a property list which already contains the ⟨*key*⟩ then the new entry overwrites the existing one. The ⟨*keys*⟩ are compared on a string basis, using the same method as `\str_if_eq:nnTF`.

Property lists are intended for storing key-based information for use within code. They can be converted from and to key–value lists, which are a form of *input* parsed by the l3keys module. If a key–value list contains a ⟨*key*⟩ multiple times, only the last ⟨*value*⟩ associated to it will be kept in the conversion to a property list.

Internally, property lists can use two distinct implementations with different data storage, which are decided when declaring the property list variable using `\prop_new:N` ("flat" storage) or `\prop_new_linked:N` ("linked" storage). After a property list is declared with `\prop_new:N` or `\prop_new_linked:N`, the type of internal data storage can be changed by `\prop_make_flat:N` or `\prop_make_linked:N`, but only at the outermost group level. All other l3prop functions transparently manipulate either storage method and convert as needed.

- The (default) "flat" storage method is suited for a relatively small number of entries, or when the property list is likely to be manipulated (copied, mapped) as a whole rather than entry-wise. It is significantly faster for `\prop_set_eq:NN`, and only slightly faster for `\prop_clear:N`, `\prop_concat:NNN`, and mapping functions `\prop_map_...`.
- The "linked" storage method is meant for property lists with a large numbers of entries. It takes up more of TeX's memory during a run, but is significantly faster (for long lists) when accessing or modifying individual entries using functions such as `\prop_if_in:Nn`, `\prop_item:Nn`, `\prop_put:Nnn`, `\prop_get:NnN`, `\prop_pop:NnN`, `\prop_remove:Nn`, as it takes a constant time for these operations (rather

than the number of items for a "flat" property list). A technical drawback is that memory is permanently used[7] by ⟨*keys*⟩ stored in a "linked" property list, even after they are removed and the property list is deleted.

25.1 Creating and initialising property lists

`\prop_new:N`
`\prop_new:c`

`\prop_new:N` ⟨*property list*⟩

Creates a new "flat" ⟨*property list*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*property list*⟩ initially contains no entries. See also `\prop_new_linked:N`.

`\prop_new_linked:N`
`\prop_new_linked:c`
New: 2024-02-12

`\prop_new_linked:N` ⟨*property list*⟩

Creates a new "linked" ⟨*property list*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*property list*⟩ initially contains no entries. The internal data storage differs from that produced by `\prop_new:N` and it is optimized for property lists with a large number of entries.

`\prop_clear:N`
`\prop_clear:c`
`\prop_gclear:N`
`\prop_gclear:c`

`\prop_clear:N` ⟨*property list*⟩

Clears all entries from the ⟨*property list*⟩.

`\prop_clear_new:N`
`\prop_clear_new:c`
`\prop_gclear_new:N`
`\prop_gclear_new:c`

`\prop_clear_new:N` ⟨*property list*⟩

Ensures that the ⟨*property list*⟩ exists globally by applying `\prop_new:N` if necessary, then applies `\prop_(g)clear:N` to leave the list empty.

TeXhackers note: If the property list exists and is of "linked" type, it is cleared but not made into a flat property list.

`\prop_clear_new_linked:N`
`\prop_clear_new_linked:c`
`\prop_gclear_new_linked:N`
`\prop_gclear_new_linked:c`
New: 2024-02-12

`\prop_clear_new_linked:N` ⟨*property list*⟩

Ensures that the ⟨*property list*⟩ exists globally by applying `\prop_new_linked:N` if necessary, then applies `\prop_(g)clear:N` to leave the list empty.

TeXhackers note: If the property list exists and is of "flat" type, it is cleared but not made into a linked property list.

`\prop_set_eq:NN`
`\prop_set_eq:(cN|Nc|cc)`
`\prop_gset_eq:NN`
`\prop_gset_eq:(cN|Nc|cc)`

`\prop_set_eq:NN` ⟨*property list*$_1$⟩ ⟨*property list*$_2$⟩

Sets the content of ⟨*property list*$_1$⟩ equal to that of ⟨*property list*$_2$⟩. This converts as needed between the two storage types.

[7] Until the end of the run, that is.

\prop_set_from_keyval:Nn
\prop_set_from_keyval:cn
\prop_gset_from_keyval:Nn
\prop_gset_from_keyval:cn

New: 2017-11-28
Updated: 2021-11-07

```
\prop_set_from_keyval:Nn ⟨property list⟩
  {
    ⟨key1⟩ = ⟨value1⟩ ,
    ⟨key2⟩ = ⟨value2⟩ , ...
  }
```

Sets ⟨*property list*⟩ to contain key–value pairs given in the second argument. If duplicate keys appear only the last of the values is kept. In contrast to most keyval lists (*e.g.* those in l3keys), each key here *must* be followed with an `=` sign even to specify an empty ⟨*value*⟩.

Spaces are trimmed around every ⟨*key*⟩ and every ⟨*value*⟩, and if the result of trimming spaces consists of a single brace group then a set of outer braces is removed. This enables both the ⟨*key*⟩ and the ⟨*value*⟩ to contain spaces, commas or equal signs. The ⟨*key*⟩ is then processed by `\tl_to_str:n`. This function correctly detects the `=` and `,` signs provided they have the standard category code 12 or they are active.

\prop_const_from_keyval:Nn
\prop_const_from_keyval:cn

New: 2017-11-28
Updated: 2021-11-07

```
\prop_const_from_keyval:Nn ⟨property list⟩
  {
    ⟨key1⟩ = ⟨value1⟩ ,
    ⟨key2⟩ = ⟨value2⟩ , ...
  }
```

Creates a new constant "flat" ⟨*property list*⟩ or raises an error if the name is already taken. The ⟨*property list*⟩ is set globally to contain key–value pairs given in the second argument, processed in the way described for `\prop_set_from_keyval:Nn`. If duplicate keys appear only the last of the values is kept. This function correctly detects the `=` and `,` signs provided they have the standard category code 12 or they are active.

\prop_const_linked_from_keyval:Nn
\prop_const_linked_from_keyval:cn

New: 2024-02-12

```
\prop_const_linked_from_keyval:Nn ⟨prop var⟩
  {
    ⟨key1⟩ = ⟨value1⟩ ,
    ⟨key2⟩ = ⟨value2⟩ , ...
  }
```

Creates a new constant "linked" ⟨*prop var*⟩ or raises an error if the name is already taken. The ⟨*prop var*⟩ is set globally to contain key–value pairs given in the second argument, processed in the way described for `\prop_set_from_keyval:Nn`. If duplicate keys appear only the last of the values is kept. This function correctly detects the `=` and `,` signs provided they have the standard category code 12 or they are active.

\prop_make_flat:N
\prop_make_flat:c

New: 2024-02-12

```
\prop_make_flat:N ⟨property list⟩
```

Changes the internal storage type of the ⟨*property list*⟩ to be the same "flat" storage as `\prop_new:N`. The key–value pairs of the ⟨*property list*⟩ are preserved by the change. If the property list was already flat then nothing is done. This function can only be used at the outermost group level.

\prop_make_linked:N
\prop_make_linked:c

New: 2024-02-12

```
\prop_make_linked:N ⟨property list⟩
```

Changes the internal storage type of the ⟨*property list*⟩ to be the same "linked" storage as `\prop_new_linked:N`. The key–value pairs of the ⟨*property list*⟩ are preserved by the change. If the property list was already linked then nothing is done. This function can only be used at the outermost group level.

25.2 Adding and updating property list entries

\prop_put:Nnn
\prop_put:(NnV|Nnv|Nne|NVn|NVV|NVv|NVe|Nvn|NvV|Nvv|Nve|Nen|NeV|Nev|Nee|Nno|Non|Noo|cnn|cnV|cnv|cne|cVn|cVV|cVv|cVe|cvn|cvV|cvv|cve|cen|ceV|cev|cee|cno|con|coo)
\prop_gput:Nnn
\prop_gput:(NnV|Nnv|Nne|NVn|NVV|NVv|NVe|Nvn|NvV|Nvv|Nve|Nen|NeV|Nev|Nee|Nno|Non|Noo|cnn|cnV|cnv|cne|cVn|cVV|cVv|cVe|cvn|cvV|cvv|cve|cen|ceV|cev|cee|cno|con|coo)

Updated: 2012-07-09

`\prop_put:Nnn` ⟨*property list*⟩ {⟨*key*⟩} {⟨*value*⟩}

Adds an entry to the ⟨*property list*⟩ which may be accessed using the ⟨*key*⟩ and which has ⟨*value*⟩. If the ⟨*key*⟩ is already present in the ⟨*property list*⟩, the existing entry is overwritten by the new ⟨*value*⟩. Both the ⟨*key*⟩ and ⟨*value*⟩ may contain any ⟨*balanced text*⟩. The ⟨*key*⟩ is stored after processing with `\tl_to_str:n`, meaning that category codes are ignored.

\prop_put_if_not_in:Nnn
\prop_put_if_not_in:(NnV|Nnv|Nne|NVn|NVV|NVv|NVe|Nvn|NvV|Nvv|Nve|Nen|NeV|Nev|Nee|cnn|cnV|cnv|cne|cVn|cVV|cVv|cVe|cvn|cvV|cvv|cve|cen|ceV|cev|cee)
\prop_gput_if_not_in:Nnn
\prop_gput_if_not_in:(NnV|Nnv|Nne|NVn|NVV|NVv|NVe|Nvn|NvV|Nvv|Nve|Nen|NeV|Nev|Nee|cnn|cnV|cnv|cne|cVn|cVV|cVv|cVe|cvn|cvV|cvv|cve|cen|ceV|cev|cee)

New: 2024-03-30
Updated: 2024-05-07

`\prop_put_if_not_in:Nnn` ⟨*property list*⟩ {⟨*key*⟩} {⟨*value*⟩}

If the ⟨*key*⟩ is present in the ⟨*property list*⟩ then no action is taken. Otherwise, a new entry is added as described for `\prop_put:Nnn`.

\prop_concat:NNN
\prop_concat:ccc
\prop_gconcat:NNN
\prop_gconcat:ccc

New: 2021-05-16

`\prop_concat:NNN` ⟨*property list*$_1$⟩ ⟨*property list*$_2$⟩ ⟨*property list3*⟩

Combines the key–value pairs of ⟨*property list*$_2$⟩ and ⟨*property list*$_3$⟩, and saves the result in ⟨*property list*$_1$⟩. If a key appears in both ⟨*property list*$_2$⟩ and ⟨*property list*$_3$⟩ then the last value, namely the value in ⟨*property list*$_3$⟩ is kept. This converts as needed between the two storage types.

\prop_put_from_keyval:Nn
\prop_put_from_keyval:cn
\prop_gput_from_keyval:Nn
\prop_gput_from_keyval:cn

New: 2021-05-16
Updated: 2021-11-07

```
\prop_put_from_keyval:Nn ⟨property list⟩
  {
    ⟨key1⟩ = ⟨value1⟩ ,
    ⟨key2⟩ = ⟨value2⟩ , ...
  }
```

Updates the ⟨*property list*⟩ by adding entries for each key–value pair given in the second argument. The addition is done through `\prop_put:Nnn`, hence if the ⟨*property list*⟩ already contains some of the keys, the corresponding values are discarded and replaced by those given in the key–value list. If duplicate keys appear in the key–value list then only the last of the values is kept.

The function is equivalent to storing the key–value pairs in a temporary property list using `\prop_set_from_keyval:Nn`, then combining ⟨*property list*⟩ with the temporary variable using `\prop_concat:NNN`. In particular, the ⟨*keys*⟩ and ⟨*values*⟩ are space-trimmed and unbraced as described in `\prop_set_from_keyval:Nn`. This function correctly detects the `=` and `,` signs provided they have the standard category code 12 or they are active.

25.3 Recovering values from property lists

\prop_get:NnN
\prop_get:(NVN|NvN|NeN|NoN|cnN|cVN|cvN|ceN|coN|cnc)

Updated: 2011-08-28

```
\prop_get:NnN ⟨property list⟩ {⟨key⟩} ⟨tl var⟩
```

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the ⟨*property list*⟩, and places this in the ⟨*token list variable*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ then the ⟨*tl var*⟩ is set to the special marker `\q_no_value`. The ⟨*token list variable*⟩ is set within the current TeX group. See also `\prop_get:NnNTF`.

\prop_pop:NnN
\prop_pop:(NVN|NoN|cnN|cVN|coN)

Updated: 2011-08-18

```
\prop_pop:NnN ⟨property list⟩ {⟨key⟩} ⟨tl var⟩
```

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the ⟨*property list*⟩, and places this in the ⟨*token list variable*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ then the ⟨*tl var*⟩ is set to the special marker `\q_no_value`. The ⟨*key*⟩ and ⟨*value*⟩ are then deleted from the property list. Both assignments are local. See also `\prop_pop:NnNTF`.

\prop_gpop:NnN
\prop_gpop:(NVN|NoN|cnN|cVN|coN)

Updated: 2011-08-18

```
\prop_gpop:NnN ⟨property list⟩ {⟨key⟩} ⟨tl var⟩
```

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the ⟨*property list*⟩, and places this in the ⟨*token list variable*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ then the ⟨*tl var*⟩ is set to the special marker `\q_no_value`. The ⟨*key*⟩ and ⟨*value*⟩ are then deleted from the property list. The ⟨*property list*⟩ is modified globally, while the assignment of the ⟨*tl var*⟩ is local. See also `\prop_gpop:NnNTF`.

`\prop_item:Nn` ⋆
`\prop_item:(NV|Ne|No|cn|cV|ce|co)` ⋆

New: 2014-07-17

`\prop_item:Nn` ⟨*property list*⟩ {⟨*key*⟩}

Expands to the ⟨*value*⟩ corresponding to the ⟨*key*⟩ in the ⟨*property list*⟩. If the ⟨*key*⟩ is missing, this has an empty expansion.

TEXhackers note: For "flat" property lists, this expandable function iterates through every key–value pair and is therefore slower than a non-expandable approach based on `\prop_get:NnN`. (For "linked" property lists both functions are fast.)

The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*value*⟩ does not expand further when appearing in an `e`-type or `x`-type argument expansion.

`\prop_count:N` ⋆
`\prop_count:c` ⋆

`\prop_count:N` ⟨*property list*⟩

Leaves the number of key–value pairs in the ⟨*property list*⟩ in the input stream as an ⟨*integer denotation*⟩.

`\prop_to_keyval:N` ⋆

`\prop_to_keyval:N` ⟨*property list*⟩

Expands to the ⟨*property list*⟩ in a key–value notation. Keep in mind that a ⟨*property list*⟩ is *unordered*, while key–value interfaces are not necessarily, so this cannot be used for arbitrary interfaces.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the key–value list does not expand further when appearing in an `e`-type or `x`-type argument expansion. It also needs exactly two steps of expansion.

25.4 Modifying property lists

`\prop_remove:Nn`
`\prop_remove:(NV|Ne|cn|cV|ce)`
`\prop_gremove:Nn`
`\prop_gremove:(NV|Ne|cn|cV|ce)`

New: 2012-05-12

`\prop_remove:Nn` ⟨*property list*⟩ {⟨*key*⟩}

Removes the entry listed under ⟨*key*⟩ from the ⟨*property list*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ no change occurs, *i.e* there is no need to test for the existence of a key before deleting it.

25.5 Property list conditionals

`\prop_if_exist_p:N` ⋆
`\prop_if_exist_p:c` ⋆
`\prop_if_exist:NTF` ⋆
`\prop_if_exist:cTF` ⋆

New: 2012-03-03

`\prop_if_exist_p:N` ⟨*property list*⟩
`\prop_if_exist:NTF` ⟨*property list*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*property list*⟩ is currently defined. This does not check that the ⟨*property list*⟩ really is a property list variable.

\prop_if_empty_p:N ⋆
\prop_if_empty_p:c ⋆
\prop_if_empty:NTF ⋆
\prop_if_empty:cTF ⋆

\prop_if_empty_p:N ⟨*property list*⟩
\prop_if_empty:NTF ⟨*property list*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*property list*⟩ is empty (containing no entries).

\prop_if_in_p:Nn ⋆
\prop_if_in_p:(NV|Ne|No|cn|cV|ce|co) ⋆
\prop_if_in:NnTF ⋆
\prop_if_in:(NV|Ne|No|cn|cV|ce|co)TF ⋆

Updated: 2011-09-15

\prop_if_in_p:Nn ⟨*property list*⟩ {⟨*key*⟩}
\prop_if_in:NnTF ⟨*property list*⟩ {⟨*key*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*key*⟩ is present in the ⟨*property list*⟩, making the comparison using the method described by \str_if_eq:nnTF.

TEXhackers note: For "flat" property lists, this expandable function iterates through every key–value pair and is therefore slower than a non-expandable approach based on \prop_get:NnNTF. (For "linked" property lists both functions are fast.)

25.6 Recovering values from property lists with branching

The functions in this section combine tests for the presence of a key in a property list with recovery of the associated valued. This makes them useful for cases where different code follows depending on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

\prop_get:NnNTF
\prop_get:(NVN|NvN|NeN|NoN|cnN|cVN|cvN|ceN|coN|cnc)TF

Updated: 2012-05-19

\prop_get:NnNTF ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩
{⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*key*⟩ is not present in the ⟨*property list*⟩, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*key*⟩ is present in the ⟨*property list*⟩, stores the corresponding ⟨*value*⟩ in the ⟨*tl var*⟩ without removing it from the ⟨*property list*⟩, then leaves the ⟨*true code*⟩ in the input stream. The ⟨*tl var*⟩ is assigned locally.

\prop_pop:NnNTF
\prop_pop:(NVN|NoN|cnN|cVN|coN)TF

New: 2011-08-18
Updated: 2012-05-19

\prop_pop:NnNTF ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩
{⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*key*⟩ is not present in the ⟨*property list*⟩, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*key*⟩ is present in the ⟨*property list*⟩, pops the corresponding ⟨*value*⟩ in the ⟨*tl var*⟩, *i.e.* removes the item from the ⟨*property list*⟩. Both the ⟨*property list*⟩ and the ⟨*tl var*⟩ are assigned locally.

\prop_gpop:NnN*TF*
\prop_gpop:(NVN|NoN|cnN|cVN|coN)*TF*

New: 2011-08-18
Updated: 2012-05-19

\prop_gpop:NnNTF ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩
{⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*key*⟩ is not present in the ⟨*property list*⟩, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*tl var*⟩ is not defined in this case and should not be relied upon. If the ⟨*key*⟩ is present in the ⟨*property list*⟩, pops the corresponding ⟨*value*⟩ in the ⟨*tl var*⟩, *i.e.* removes the item from the ⟨*property list*⟩. The ⟨*property list*⟩ is modified globally, while the ⟨*tl var*⟩ is assigned locally.

25.7 Mapping over property lists

All mappings are done at the current group level, *i.e.* any local assignments made by the ⟨*function*⟩ or ⟨*code*⟩ discussed below remain in effect after the loop.

\prop_map_function:NN ☆
\prop_map_function:cN ☆

Updated: 2013-01-08

\prop_map_function:NN ⟨*property list*⟩ ⟨*function*⟩

Applies ⟨*function*⟩ to every ⟨*entry*⟩ stored in the ⟨*property list*⟩. The ⟨*function*⟩ receives two arguments for each iteration: the ⟨*key*⟩ and associated ⟨*value*⟩. The order in which ⟨*entries*⟩ are returned is not defined and should not be relied upon. To pass further arguments to the ⟨*function*⟩, see \prop_map_inline:Nn (non-expandable) or \prop_map_tokens:Nn.

\prop_map_inline:Nn
\prop_map_inline:cn

Updated: 2013-01-08

\prop_map_inline:Nn ⟨*property list*⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ to every ⟨*entry*⟩ stored within the ⟨*property list*⟩. The ⟨*inline function*⟩ should consist of code which receives the ⟨*key*⟩ as #1 and the ⟨*value*⟩ as #2. The order in which ⟨*entries*⟩ are returned is not defined and should not be relied upon.

\prop_map_tokens:Nn ☆
\prop_map_tokens:cn ☆

\prop_map_tokens:Nn ⟨*property list*⟩ {⟨*code*⟩}

Analogue of \prop_map_function:NN which maps several tokens instead of a single function. The ⟨*code*⟩ receives each key–value pair in the ⟨*property list*⟩ as two trailing brace groups. For instance,

```
\prop_map_tokens:Nn \l_my_prop { \str_if_eq:nnT { mykey } }
```

expands to the value corresponding to mykey: for each pair in \l_my_prop the function \str_if_eq:nnT receives mykey, the ⟨*key*⟩ and the ⟨*value*⟩ as its three arguments. For that specific task, \prop_item:Nn is faster.

\prop_map_break: ☆ \prop_map_break:

Updated: 2012-06-29

Used to terminate a \prop_map_... function before all entries in the ⟨*property list*⟩ have been processed. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \prop_map_break: }
      {
        % Do something useful
      }
  }
```

Use outside of a \prop_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

\prop_map_break:n ☆ \prop_map_break:n {⟨*code*⟩}

Updated: 2012-06-29

Used to terminate a \prop_map_... function before all entries in the ⟨*property list*⟩ have been processed, inserting the ⟨*code*⟩ after the mapping has ended. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \prop_map_break:n { <code> } }
      {
        % Do something useful
      }
  }
```

Use outside of a \prop_map_... scenario leads to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted before the ⟨*code*⟩ is inserted into the input stream. This depends on the design of the mapping function.

25.8 Viewing property lists

\prop_show:N
\prop_show:c

Updated: 2021-04-29

\prop_show:N ⟨*property list*⟩

Displays the entries in the ⟨*property list*⟩ in the terminal, and specifies its storage type.

\prop_log:N
\prop_log:c

New: 2014-08-12
Updated: 2021-04-29

\prop_log:N ⟨*property list*⟩

Writes the entries in the ⟨*property list*⟩ in the log file, and specifies its storage type.

25.9 Scratch property lists

There is no need to include both flat and linked property lists as scratch variables. We arbitrarily pick the older implementation.

\l_tmpa_prop
\l_tmpb_prop

New: 2012-06-23

Scratch "flat" property lists for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_prop
\g_tmpb_prop

New: 2012-06-23

Scratch "flat" property lists for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

25.10 Constants

\c_empty_prop

A permanently-empty property list used for internal comparisons.

Chapter 26

The l3skip module Dimensions and skips

LaTeX3 provides two general length variables: `dim` and `skip`. Lengths stored as `dim` variables have a fixed length, whereas `skip` lengths have a rubber (stretch/shrink) component. In addition, the `muskip` type is available for use in math mode: this is a special form of `skip` where the lengths involved are determined by the current math font (in `mu`). There are common features in the creation and setting of length variables, but for clarity the functions are grouped by variable type.

Many functions take *dimension expressions* ("⟨*dim expr*⟩") or *skip expressions* ("⟨*skip expr*⟩") as arguments.

26.1 Creating and initialising dim variables

`\dim_new:N`
`\dim_new:c`

`\dim_new:N ⟨dimension⟩`

Creates a new ⟨*dimension*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*dimension*⟩ is initially equal to 0 pt.

`\dim_const:Nn`
`\dim_const:cn`

New: 2012-03-05

`\dim_const:Nn ⟨dimension⟩ {⟨dim expr⟩}`

Creates a new constant ⟨*dimension*⟩ or raises an error if the name is already taken. The value of the ⟨*dimension*⟩ is set globally to the ⟨*dim expr*⟩.

`\dim_zero:N`
`\dim_zero:c`
`\dim_gzero:N`
`\dim_gzero:c`

`\dim_zero:N ⟨dimension⟩`

Sets ⟨*dimension*⟩ to 0 pt.

`\dim_zero_new:N`
`\dim_zero_new:c`
`\dim_gzero_new:N`
`\dim_gzero_new:c`

New: 2012-01-07

`\dim_zero_new:N ⟨dimension⟩`

Ensures that the ⟨*dimension*⟩ exists globally by applying `\dim_new:N` if necessary, then applies `\dim_(g)zero:N` to leave the ⟨*dimension*⟩ set to zero.

\dim_if_exist_p:N ⋆
\dim_if_exist_p:c ⋆
\dim_if_exist:NTF ⋆
\dim_if_exist:cTF ⋆

New: 2012-03-03

\dim_if_exist_p:N ⟨dimension⟩
\dim_if_exist:NTF ⟨dimension⟩ {⟨true code⟩} {⟨false code⟩}

Tests whether the ⟨dimension⟩ is currently defined. This does not check that the ⟨dimension⟩ really is a dimension variable.

26.2 Setting dim variables

\dim_add:Nn
\dim_add:cn
\dim_gadd:Nn
\dim_gadd:cn

Updated: 2011-10-22

\dim_add:Nn ⟨dimension⟩ {⟨dim expr⟩}

Adds the result of the ⟨dim expr⟩ to the current content of the ⟨dimension⟩.

\dim_set:Nn
\dim_set:(cn|NV|cV)
\dim_gset:Nn
\dim_gset:(cn|NV|cV)

Updated: 2011-10-22

\dim_set:Nn ⟨dimension⟩ {⟨dim expr⟩}

Sets ⟨dimension⟩ to the value of ⟨dim expr⟩, which must evaluate to a length with units.

\dim_set_eq:NN
\dim_set_eq:(cN|Nc|cc)
\dim_gset_eq:NN
\dim_gset_eq:(cN|Nc|cc)

\dim_set_eq:NN ⟨dimension$_1$⟩ ⟨dimension$_2$⟩

Sets the content of ⟨dimension$_1$⟩ equal to that of ⟨dimension$_2$⟩.

\dim_sub:Nn
\dim_sub:cn
\dim_gsub:Nn
\dim_gsub:cn

Updated: 2011-10-22

\dim_sub:Nn ⟨dimension⟩ {⟨dim expr⟩}

Subtracts the result of the ⟨dim expr⟩ from the current content of the ⟨dimension⟩.

26.3 Utilities for dimension calculations

\dim_abs:n ⋆

Updated: 2012-09-26

\dim_abs:n {⟨dim expr⟩}

Converts the ⟨dim expr⟩ to its absolute value, leaving the result in the input stream as a ⟨dimension denotation⟩.

\dim_max:nn ⋆
\dim_min:nn ⋆

New: 2012-09-09
Updated: 2012-09-26

\dim_max:nn {⟨dim expr$_1$⟩} {⟨dim expr$_2$⟩}
\dim_min:nn {⟨dim expr$_1$⟩} {⟨dim expr$_2$⟩}

Evaluates the two ⟨dim exprs⟩ and leaves either the maximum or minimum value in the input stream as appropriate, as a ⟨dimension denotation⟩.

\dim_ratio:nn ☆ `\dim_ratio:nn {⟨dim expr₁⟩} {⟨dim expr₂⟩}`

Updated: 2011-10-22

Parses the two ⟨*dim exprs*⟩ and converts the ratio of the two to a form suitable for use inside a ⟨*dim expr*⟩. This ratio is then left in the input stream, allowing syntax such as

```
\dim_set:Nn \l_my_dim
  { 10 pt * \dim_ratio:nn { 5 pt } { 10 pt } }
```

The output of `\dim_ratio:nn` on full expansion is a ratio expression between two integers, with all distances converted to scaled points. Thus

```
\tl_set:Ne \l_my_tl { \dim_ratio:nn { 5 pt } { 10 pt } }
\tl_show:N \l_my_tl
```

displays `327680/655360` on the terminal.

26.4 Dimension expression conditionals

\dim_compare_p:nNn ★
\dim_compare:nNn*TF* ★

```
\dim_compare_p:nNn {⟨dim expr₁⟩} ⟨relation⟩ {⟨dim expr₂⟩}
\dim_compare:nNnTF
  {⟨dim expr₁⟩} ⟨relation⟩ {⟨dim expr₂⟩}
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates each of the ⟨*dim exprs*⟩ as described for `\dim_eval:n`. The two results are then compared using the ⟨*relation*⟩:

Equal	=
Greater than	>
Less than	<

This function is less flexible than `\dim_compare:nTF` but around 5 times faster.

`\dim_compare_p:n` ⋆
`\dim_compare:nTF` ⋆

Updated: 2013-01-13

```
\dim_compare_p:n
  {
    ⟨dim expr1⟩ ⟨relation1⟩
    ...
    ⟨dim exprN⟩ ⟨relationN⟩
    ⟨dim exprN+1⟩
  }
\dim_compare:nTF
  {
    ⟨dim expr1⟩ ⟨relation1⟩
    ...
    ⟨dim exprN⟩ ⟨relationN⟩
    ⟨dim exprN+1⟩
  }
  {⟨true code⟩} {⟨false code⟩}
```

This function evaluates the ⟨*dim exprs*⟩ as described for `\dim_eval:n` and compares consecutive result using the corresponding ⟨*relation*⟩, namely it compares ⟨*dim expr*$_1$⟩ and ⟨*dim expr*$_2$⟩ using the ⟨*relation*$_1$⟩, then ⟨*dim expr*$_2$⟩ and ⟨*dim expr*$_3$⟩ using the ⟨*relation*$_2$⟩, until finally comparing ⟨*dim expr*$_N$⟩ and ⟨*dim expr*$_{N+1}$⟩ using the ⟨*relation*$_N$⟩. The test yields `true` if all comparisons are `true`. Each ⟨*dim expr*⟩ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other ⟨*dim expr*⟩ is evaluated and no other comparison is performed. The ⟨*relations*⟩ can be any of the following:

Equal	`=` or `==`
Greater than or equal to	`>=`
Greater than	`>`
Less than or equal to	`<=`
Less than	`<`
Not equal	`!=`

This function is more flexible than `\dim_compare:nNnTF` but around 5 times slower.

`\dim_case:nn` ★
`\dim_case:nnTF` ★

New: 2013-07-24

```
\dim_case:nnTF {⟨test dim expr⟩}
  {
    {⟨dim expr case1⟩} {⟨code case1⟩}
    {⟨dim expr case2⟩} {⟨code case2⟩}
    ...
    {⟨dim expr casen⟩} {⟨code casen⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function evaluates the ⟨*test dim expr*⟩ and compares this in turn to each of the ⟨*dim expr case*⟩s until a match is found. If the two are equal then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function `\dim_case:nn`, which does nothing if there is no match, is also available. For example

```
\dim_set:Nn \l_tmpa_dim { 5 pt }
\dim_case:nnF
  { 2 \l_tmpa_dim }
  {
    { 5 pt }        { Small }
    { 4 pt + 6 pt } { Medium }
    { - 10 pt }     { Negative }
  }
  { No idea! }
```

leaves "`Medium`" in the input stream. Since evaluation of the test expressions stops at the first successful case, the order of possible matches should normally be that the most likely are earlier: this will reduce the average steps required to complete expansion.

26.5 Dimension expression loops

`\dim_do_until:nNnn` ☆

```
\dim_do_until:nNnn {⟨dim expr1⟩} ⟨relation⟩ {⟨dim expr2⟩} {⟨code⟩}
```

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*dim exprs*⟩ as described for `\dim_compare:nNnTF`. If the test is `false` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `true`.

`\dim_do_while:nNnn` ☆

```
\dim_do_while:nNnn {⟨dim expr1⟩} ⟨relation⟩ {⟨dim expr2⟩} {⟨code⟩}
```

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*dim exprs*⟩ as described for `\dim_compare:nNnTF`. If the test is `true` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `false`.

`\dim_until_do:nNnn` ☆ `\dim_until_do:nNnn {⟨dim expr₁⟩} ⟨relation⟩ {⟨dim expr₂⟩} {⟨code⟩}`

Evaluates the relationship between the two ⟨*dim exprs*⟩ as described for `\dim_compare:nNnTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `false`. After the ⟨*code*⟩ has been processed by TeX the test is repeated, and a loop occurs until the test is `true`.

`\dim_while_do:nNnn` ☆ `\dim_while_do:nNnn {⟨dim expr₁⟩} ⟨relation⟩ {⟨dim expr₂⟩} {⟨code⟩}`

Evaluates the relationship between the two ⟨*dim exprs*⟩ as described for `\dim_compare:nNnTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `true`. After the ⟨*code*⟩ has been processed by TeX the test is repeated, and a loop occurs until the test is `false`.

`\dim_do_until:nn` ☆ `\dim_do_until:nn {⟨dimension relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the ⟨*dimension relation*⟩ as described for `\dim_compare:nTF`. If the test is `false` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `true`.

`\dim_do_while:nn` ☆ `\dim_do_while:nn {⟨dimension relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the ⟨*dimension relation*⟩ as described for `\dim_compare:nTF`. If the test is `true` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `false`.

`\dim_until_do:nn` ☆ `\dim_until_do:nn {⟨dimension relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Evaluates the ⟨*dimension relation*⟩ as described for `\dim_compare:nTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `false`. After the ⟨*code*⟩ has been processed by TeX the test is repeated, and a loop occurs until the test is `true`.

`\dim_while_do:nn` ☆ `\dim_while_do:nn {⟨dimension relation⟩} {⟨code⟩}`

Updated: 2013-01-13

Evaluates the ⟨*dimension relation*⟩ as described for `\dim_compare:nTF`, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is `true`. After the ⟨*code*⟩ has been processed by TeX the test is repeated, and a loop occurs until the test is `false`.

26.6 Dimension step functions

`\dim_step_function:nnnN` ☆ `\dim_step_function:nnnN {⟨initial value⟩} {⟨step⟩} {⟨final value⟩} ⟨function⟩`

New: 2018-02-18

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be dimension expressions. The ⟨*function*⟩ is then placed in front of each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩). The ⟨*step*⟩ must be non-zero. If the ⟨*step*⟩ is positive, the loop stops when the ⟨*value*⟩ becomes larger than the ⟨*final value*⟩. If the ⟨*step*⟩ is negative, the loop stops when the ⟨*value*⟩ becomes smaller than the ⟨*final value*⟩. The ⟨*function*⟩ should absorb one argument.

\dim_step_inline:nnnn

New: 2018-02-18

\dim_step_inline:nnnn {⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} {⟨*code*⟩}

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be dimension expressions. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream with #1 replaced by the current ⟨*value*⟩. Thus the ⟨*code*⟩ should define a function of one argument (#1).

\dim_step_variable:nnnNn

New: 2018-02-18

\dim_step_variable:nnnNn
{⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} ⟨*tl var*⟩ {⟨*code*⟩}

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be dimension expressions. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream, with the ⟨*tl var*⟩ defined as the current ⟨*value*⟩. Thus the ⟨*code*⟩ should make use of the ⟨*tl var*⟩.

26.7 Using dim expressions and variables

\dim_eval:n ⋆

Updated: 2011-10-22

\dim_eval:n {⟨*dim expr*⟩}

Evaluates the ⟨*dim expr*⟩, expanding any dimensions and token list variables within the ⟨*expression*⟩ to their content (without requiring \dim_use:N/\tl_use:N) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a ⟨*dimension denotation*⟩ after two expansions. This is expressed in points (pt), and requires suitable termination if used in a TeX-style assignment as it is *not* an ⟨*internal dimension*⟩.

\dim_sign:n ⋆

New: 2018-11-03

\dim_sign:n {⟨*dim expr*⟩}

Evaluates the ⟨*dim expr*⟩ then leaves 1 or 0 or −1 in the input stream according to the sign of the result.

\dim_use:N ⋆
\dim_use:c ⋆

\dim_use:N ⟨*dimension*⟩

Recovers the content of a ⟨*dimension*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a ⟨*dimension*⟩ is required (such as in the argument of \dim_eval:n).

TeXhackers note: \dim_use:N is the TeX primitive \the: this is one of several LaTeX3 names for this primitive.

\dim_to_decimal:n ⋆

New: 2014-07-15

\dim_to_decimal:n {⟨*dim expr*⟩}

Evaluates the ⟨*dim expr*⟩, and leaves the result, expressed in points (pt) in the input stream, with *no units*. The result is rounded by TeX to at most five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal:n { 1bp }
```

leaves 1.00374 in the input stream, *i.e.* the magnitude of one "big point" when converted to (TeX) points.

`\dim_to_decimal_in_bp:n` ⋆

New: 2014-07-15
Updated: 2023-05-20

`\dim_to_decimal_in_bp:n {⟨dim expr⟩}`

Evaluates the ⟨*dim expr*⟩, and leaves the result, expressed in big points (`bp`) in the input stream, with *no units*. The result is rounded by TEX to at most five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_bp:n { 1pt }
```

leaves `0.99628` in the input stream, *i.e.* the magnitude of one (TEX) point when converted to big points.

TEXhackers note: The implementation of this function is re-entrant: the result of

```
\dim_compare:nNnTF
  { <x>bp } =
    { \dim_to_decimal_in_bp:n { <x>bp } bp }
```

will be logically `true`. The decimal representations may differ provided they produce the same TEX dimension.

`\dim_to_decimal_in_cc:n` ⋆
`\dim_to_decimal_in_cm:n` ⋆
`\dim_to_decimal_in_dd:n` ⋆
`\dim_to_decimal_in_in:n` ⋆
`\dim_to_decimal_in_mm:n` ⋆
`\dim_to_decimal_in_pc:n` ⋆

New: 2023-05-20

`\dim_to_decimal_in_cm:n {⟨dim expr⟩}`

Evaluates the ⟨*dim expr*⟩, and leaves the result, expressed with the appropriate scaling in the input stream, with *no units*. If the decimal part of the result is zero, it is omitted, together with the decimal marker. The precisions of the result is limited to a maximum of five decimal places with trailing zeros omitted.

The maximum TEX allowable dimension value (available as `\maxdimen` in plain TEX and LATEX and `\c_max_dim` in expl3) can only be expressed exactly in the units `pt`, `bp` and `sp`. The maximum allowable input values to five decimal places are

$$\begin{array}{r}1276.00215\,\mathrm{cc}\\575.83174\,\mathrm{cm}\\15312.02584\,\mathrm{dd}\\226.70540\,\mathrm{in}\\5758.31742\,\mathrm{mm}\\1365.33333\,\mathrm{pc}\end{array}$$

(Note that these are not all equal, but rather any larger value will overflow due to the way TEX converts to `sp`.) Values given to five decimal places larger that these will result in TEX errors; the behavior if additional decimal places are given depends on the TEX internals and thus larger values are *not* supported by expl3.

TEXhackers note: The implementation of these functions is re-entrant: the result of

```
\dim_compare:nNnTF
  { <x><unit> } =
    { \dim_to_decimal_in_<unit>:n { <x><unit> } <unit> }
```

will be logically `true`. The decimal representations may differ provided they produce the same TEX dimension.

`\dim_to_decimal_in_sp:n` ⋆ `\dim_to_decimal_in_sp:n {`⟨*dim expr*⟩`}`

New: 2015-05-18

Evaluates the ⟨*dim expr*⟩, and leaves the result, expressed in scaled points (`sp`) in the input stream, with *no units*. The result is necessarily an integer.

`\dim_to_decimal_in_unit:nn` ⋆ `\dim_to_decimal_in_unit:nn {`⟨*dim expr*$_1$⟩`} {`⟨*dim expr*$_2$⟩`}`

New: 2014-07-15
Updated: 2023-05-20

Evaluates the ⟨*dim exprs*⟩, and leaves the value of ⟨*dim expr*$_1$⟩, expressed in a unit given by ⟨*dim expr*$_2$⟩, in the input stream. If the decimal part of the result is zero, it is omitted, together with the decimal marker. The precisions of the result is limited to a maximum of five decimal places with trailing zeros omitted.

For example

```
\dim_to_decimal_in_unit:nn { 1bp } { 1mm }
```

leaves `0.35278` in the input stream, *i.e.* the magnitude of one big point when expressed in millimetres. The conversions do *not* guarantee that TeX would yield identical results for the direct input in an equality test, thus for instance

```
\dim_compare:nNnTF
  { 1bp } =
    { \dim_to_decimal_in_unit:nn { 1bp } { 1mm } mm }
```

will take the `false` branch.

`\dim_to_fp:n` ⋆ `\dim_to_fp:n {`⟨*dim expr*⟩`}`

New: 2012-05-08

Expands to an internal floating point number equal to the value of the ⟨*dim expr*⟩ in `pt`. Since dimension expressions are evaluated much faster than their floating point equivalent, `\dim_to_fp:n` can be used to speed up parts of a computation where a low precision and a smaller range are acceptable.

26.8 Viewing dim variables

`\dim_show:N`
`\dim_show:c`

`\dim_show:N` ⟨*dimension*⟩

Displays the value of the ⟨*dimension*⟩ on the terminal.

`\dim_show:n` `\dim_show:n {`⟨*dim expr*⟩`}`

New: 2011-11-22
Updated: 2015-08-07

Displays the result of evaluating the ⟨*dim expr*⟩ on the terminal.

`\dim_log:N`
`\dim_log:c`

New: 2014-08-22
Updated: 2015-08-03

`\dim_log:N` ⟨*dimension*⟩

Writes the value of the ⟨*dimension*⟩ in the log file.

\dim_log:n

\dim_log:n {⟨*dim expr*⟩}

New: 2014-08-22
Updated: 2015-08-07

Writes the result of evaluating the ⟨*dim expr*⟩ in the log file.

26.9 Constant dimensions

\c_max_dim

The maximum value that can be stored as a dimension. This can also be used as a component of a skip.

\c_zero_dim

A zero length as a dimension. This can also be used as a component of a skip.

26.10 Scratch dimensions

\l_tmpa_dim
\l_tmpb_dim

Scratch dimension for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_dim
\g_tmpb_dim

Scratch dimension for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

26.11 Creating and initialising skip variables

\skip_new:N
\skip_new:c

\skip_new:N ⟨*skip*⟩

Creates a new ⟨*skip*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*skip*⟩ is initially equal to 0 pt.

\skip_const:Nn
\skip_const:cn

New: 2012-03-05

\skip_const:Nn ⟨*skip*⟩ {⟨*skip expr*⟩}

Creates a new constant ⟨*skip*⟩ or raises an error if the name is already taken. The value of the ⟨*skip*⟩ is set globally to the ⟨*skip expr*⟩.

\skip_zero:N
\skip_zero:c
\skip_gzero:N
\skip_gzero:c

\skip_zero:N ⟨*skip*⟩

Sets ⟨*skip*⟩ to 0 pt.

\skip_zero_new:N
\skip_zero_new:c
\skip_gzero_new:N
\skip_gzero_new:c

New: 2012-01-07

\skip_zero_new:N ⟨*skip*⟩

Ensures that the ⟨*skip*⟩ exists globally by applying \skip_new:N if necessary, then applies \skip_(g)zero:N to leave the ⟨*skip*⟩ set to zero.

\skip_if_exist_p:N ⋆
\skip_if_exist_p:c ⋆
\skip_if_exist:NTF ⋆
\skip_if_exist:cTF ⋆

New: 2012-03-03

\skip_if_exist_p:N ⟨*skip*⟩
\skip_if_exist:NTF ⟨*skip*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*skip*⟩ is currently defined. This does not check that the ⟨*skip*⟩ really is a skip variable.

26.12 Setting skip variables

\skip_add:Nn
\skip_add:cn
\skip_gadd:Nn
\skip_gadd:cn

Updated: 2011-10-22

\skip_add:Nn ⟨*skip*⟩ {⟨*skip expr*⟩}

Adds the result of the ⟨*skip expr*⟩ to the current content of the ⟨*skip*⟩.

\skip_set:Nn
\skip_set:(cn|NV|cV)
\skip_gset:Nn
\skip_gset:(cn|NV|cV)

Updated: 2011-10-22

\skip_set:Nn ⟨*skip*⟩ {⟨*skip expr*⟩}

Sets ⟨*skip*⟩ to the value of ⟨*skip expr*⟩, which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm.

\skip_set_eq:NN
\skip_set_eq:(cN|Nc|cc)
\skip_gset_eq:NN
\skip_gset_eq:(cN|Nc|cc)

\skip_set_eq:NN ⟨*skip*$_1$⟩ ⟨*skip*$_2$⟩

Sets the content of ⟨*skip*$_1$⟩ equal to that of ⟨*skip*$_2$⟩.

\skip_sub:Nn
\skip_sub:cn
\skip_gsub:Nn
\skip_gsub:cn

Updated: 2011-10-22

\skip_sub:Nn ⟨*skip*⟩ {⟨*skip expr*⟩}

Subtracts the result of the ⟨*skip expr*⟩ from the current content of the ⟨*skip*⟩.

26.13 Skip expression conditionals

`\skip_if_eq_p:nn` ⋆
`\skip_if_eq:nnTF` ⋆

`\skip_if_eq_p:nn {⟨skip expr1⟩} {⟨skip expr2⟩}`
`\skip_if_eq:nnTF`
`{⟨skip expr1⟩} {⟨skip expr2⟩}`
`{⟨true code⟩} {⟨false code⟩}`

This function first evaluates each of the ⟨*skip exprs*⟩ as described for `\skip_eval:n`. The two results are then compared for exact equality, *i.e.* both the fixed and rubber components must be the same for the test to be true.

`\skip_if_finite_p:n` ⋆
`\skip_if_finite:nTF` ⋆
New: 2012-03-05

`\skip_if_finite_p:n {⟨skip expr⟩}`
`\skip_if_finite:nTF {⟨skip expr⟩} {⟨true code⟩} {⟨false code⟩}`

Evaluates the ⟨*skip expr*⟩ as described for `\skip_eval:n`, and then tests if all of its components are finite.

26.14 Using skip expressions and variables

`\skip_eval:n` ⋆
Updated: 2011-10-22

`\skip_eval:n {⟨skip expr⟩}`

Evaluates the ⟨*skip expr*⟩, expanding any skips and token list variables within the ⟨*expression*⟩ to their content (without requiring `\skip_use:N`/`\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a ⟨*glue denotation*⟩ after two expansions. This is expressed in points (`pt`), and requires suitable termination if used in a TeX-style assignment as it is *not* an ⟨*internal glue*⟩.

`\skip_use:N` ⋆
`\skip_use:c` ⋆

`\skip_use:N ⟨skip⟩`

Recovers the content of a ⟨*skip*⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a ⟨*dimension*⟩ or ⟨*skip*⟩ is required (such as in the argument of `\skip_eval:n`).

TeXhackers note: `\skip_use:N` is the TeX primitive `\the`: this is one of several LaTeX3 names for this primitive.

26.15 Viewing skip variables

`\skip_show:N`
`\skip_show:c`
Updated: 2015-08-03

`\skip_show:N ⟨skip⟩`

Displays the value of the ⟨*skip*⟩ on the terminal.

`\skip_show:n`
New: 2011-11-22
Updated: 2015-08-07

`\skip_show:n {⟨skip expr⟩}`

Displays the result of evaluating the ⟨*skip expr*⟩ on the terminal.

\skip_log:N
\skip_log:c

New: 2014-08-22
Updated: 2015-08-03

\skip_log:N ⟨*skip*⟩

Writes the value of the ⟨*skip*⟩ in the log file.

\skip_log:n

New: 2014-08-22
Updated: 2015-08-07

\skip_log:n {⟨*skip expr*⟩}

Writes the result of evaluating the ⟨*skip expr*⟩ in the log file.

26.16 Constant skips

\c_max_skip

Updated: 2012-11-02

The maximum value that can be stored as a skip (equal to \c_max_dim in length), with no stretch nor shrink component.

\c_zero_skip

Updated: 2012-11-01

A zero length as a skip, with no stretch nor shrink component.

26.17 Scratch skips

\l_tmpa_skip
\l_tmpb_skip

Scratch skip for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_skip
\g_tmpb_skip

Scratch skip for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

26.18 Inserting skips into the output

\skip_horizontal:N
\skip_horizontal:c
\skip_horizontal:n

Updated: 2011-10-22

\skip_horizontal:N ⟨*skip*⟩
\skip_horizontal:n {⟨*skip expr*⟩}

Inserts a horizontal ⟨*skip*⟩ into the current list. The argument can also be a ⟨*dim*⟩.

TeXhackers note: \skip_horizontal:N is the TeX primitive \hskip.

\skip_vertical:N
\skip_vertical:c
\skip_vertical:n

Updated: 2011-10-22

\skip_vertical:N ⟨*skip*⟩
\skip_vertical:n {⟨*skip expr*⟩}

Inserts a vertical ⟨*skip*⟩ into the current list. The argument can also be a ⟨*dim*⟩.

> **TeXhackers note:** \skip_vertical:N is the TeX primitive \vskip.

26.19 Creating and initialising muskip variables

\muskip_new:N
\muskip_new:c

\muskip_new:N ⟨*muskip*⟩

Creates a new ⟨*muskip*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*muskip*⟩ is initially equal to 0 mu.

\muskip_const:Nn
\muskip_const:cn

New: 2012-03-05

\muskip_const:Nn ⟨*muskip*⟩ {⟨*muskip expr*⟩}

Creates a new constant ⟨*muskip*⟩ or raises an error if the name is already taken. The value of the ⟨*muskip*⟩ is set globally to the ⟨*muskip expr*⟩.

\muskip_zero:N
\muskip_zero:c
\muskip_gzero:N
\muskip_gzero:c

\skip_zero:N ⟨*muskip*⟩

Sets ⟨*muskip*⟩ to 0 mu.

\muskip_zero_new:N
\muskip_zero_new:c
\muskip_gzero_new:N
\muskip_gzero_new:c

New: 2012-01-07

\muskip_zero_new:N ⟨*muskip*⟩

Ensures that the ⟨*muskip*⟩ exists globally by applying \muskip_new:N if necessary, then applies \muskip_(g)zero:N to leave the ⟨*muskip*⟩ set to zero.

\muskip_if_exist_p:N ⋆
\muskip_if_exist_p:c ⋆
\muskip_if_exist:N*TF* ⋆
\muskip_if_exist:c*TF* ⋆

New: 2012-03-03

\muskip_if_exist_p:N ⟨*muskip*⟩
\muskip_if_exist:NTF ⟨*muskip*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*muskip*⟩ is currently defined. This does not check that the ⟨*muskip*⟩ really is a muskip variable.

26.20 Setting muskip variables

\muskip_add:Nn
\muskip_add:cn
\muskip_gadd:Nn
\muskip_gadd:cn

Updated: 2011-10-22

\muskip_add:Nn ⟨*muskip*⟩ {⟨*muskip expr*⟩}

Adds the result of the ⟨*muskip expr*⟩ to the current content of the ⟨*muskip*⟩.

\muskip_set:Nn
\muskip_set:(cn|Nn|cn)
\muskip_gset:Nn
\muskip_gset:(cn|NV|cV)

Updated: 2011-10-22

\muskip_set:Nn ⟨*muskip*⟩ {⟨*muskip expr*⟩}

Sets ⟨muskip⟩ to the value of ⟨muskip expr⟩, which must evaluate to a math length with units and may include a rubber component (for example `1 mu plus 0.5 mu`.

\muskip_set_eq:NN
\muskip_set_eq:(cN|Nc|cc)
\muskip_gset_eq:NN
\muskip_gset_eq:(cN|Nc|cc)

\muskip_set_eq:NN ⟨*muskip*$_1$⟩ ⟨*muskip*$_2$⟩

Sets the content of ⟨muskip$_1$⟩ equal to that of ⟨muskip$_2$⟩.

\muskip_sub:Nn
\muskip_sub:cn
\muskip_gsub:Nn
\muskip_gsub:cn

Updated: 2011-10-22

\muskip_sub:Nn ⟨*muskip*⟩ {⟨*muskip expr*⟩}

Subtracts the result of the ⟨muskip expr⟩ from the current content of the ⟨muskip⟩.

26.21 Using muskip expressions and variables

\muskip_eval:n ⋆

Updated: 2011-10-22

\muskip_eval:n {⟨*muskip expr*⟩}

Evaluates the ⟨muskip expr⟩, expanding any skips and token list variables within the ⟨expression⟩ to their content (without requiring \muskip_use:N/\tl_use:N) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a ⟨muglue denotation⟩ after two expansions. This is expressed in `mu`, and requires suitable termination if used in a TeX-style assignment as it is *not* an ⟨internal muglue⟩.

\muskip_use:N ⋆
\muskip_use:c ⋆

\muskip_use:N ⟨*muskip*⟩

Recovers the content of a ⟨skip⟩ and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a ⟨dimension⟩ is required (such as in the argument of \muskip_eval:n).

TeXhackers note: \muskip_use:N is the TeX primitive \the: this is one of several LaTeX3 names for this primitive.

26.22 Viewing muskip variables

\muskip_show:N
\muskip_show:c

Updated: 2015-08-03

\muskip_show:N ⟨*muskip*⟩

Displays the value of the ⟨muskip⟩ on the terminal.

`\muskip_show:n` `\muskip_show:n {⟨muskip expr⟩}`

New: 2011-11-22
Updated: 2015-08-07

Displays the result of evaluating the ⟨*muskip expr*⟩ on the terminal.

`\muskip_log:N` `\muskip_log:N ⟨muskip⟩`
`\muskip_log:c`

New: 2014-08-22
Updated: 2015-08-03

Writes the value of the ⟨*muskip*⟩ in the log file.

`\muskip_log:n` `\muskip_log:n {⟨muskip expr⟩}`

New: 2014-08-22
Updated: 2015-08-07

Writes the result of evaluating the ⟨*muskip expr*⟩ in the log file.

26.23 Constant muskips

`\c_max_muskip` The maximum value that can be stored as a muskip, with no stretch nor shrink component.

`\c_zero_muskip` A zero length as a muskip, with no stretch nor shrink component.

26.24 Scratch muskips

`\l_tmpa_muskip`
`\l_tmpb_muskip`

Scratch muskip for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_muskip`
`\g_tmpb_muskip`

Scratch muskip for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

26.25 Primitive conditional

`\if_dim:w` ⋆

```
\if_dim:w ⟨dimen1⟩ ⟨relation⟩ ⟨dimen2⟩
  ⟨true code⟩
\else:
  ⟨false⟩
\fi:
```

Compare two dimensions. The ⟨*relation*⟩ is one of `<`, `=` or `>` with category code 12.

> **TeXhackers note:** This is the TeX primitive `\ifdim`.

Chapter 27

The l3keys module
Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. The system normally results in input of the form

```
\MyModuleSetup{
  key-one = value one,
  key-two = value two
}
```

or

```
\MyModuleMacro[
  key-one = value one,
  key-two = value two
]{argument}
```

for the user.

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimising the number of functions and arguments required. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { mymodule }
  {
    key-one .code:n   = code including parameter #1,
    key-two .tl_set:N = \l_mymodule_store_tl
  }
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { mymodule }
  {
    key-one = value one,
    key-two = value two
  }
```

As illustrated, keys are created inside a ⟨*module*⟩: a set of related keys, typically those for a single module/LaTeX 2ε package. See Section 27.2 for suggestions on how to divide large numbers of keys for a single module.

At a document level, `\keys_set:nn` is used within a document function, for example

```
\DeclareDocumentCommand \MyModuleSetup { m }
  { \keys_set:nn { mymodule } { #1 }  }
\DeclareDocumentCommand \MyModuleMacro { o m }
  {
    \group_begin:
      \keys_set:nn { mymodule } { #1 }
      % Main code for \MyModuleMacro
    \group_end:
  }
```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`. As discussed in section 27.2, it is suggested that the character `/` is reserved for sub-division of keys into different subsets. Functions and variables are *not* expanded when creating key names, and so

```
\tl_set:Nn \l_mymodule_tmp_tl { key }
\keys_define:nn { mymodule }
  {
    \l_mymodule_tmp_tl .code:n = code
  }
```

creates a key called `\l_mymodule_tmp_tl`, and not one called `key`.

27.1 Creating keys

`\keys_define:nn`
`\keys_define:ne`

Updated: 2017-11-14

`\keys_define:nn {⟨module⟩} {⟨keyval list⟩}`

Parses the ⟨*keyval list*⟩ and defines the keys listed there for ⟨*module*⟩. The ⟨*module*⟩ name is treated as a string. In practice the ⟨*module*⟩ should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The ⟨*keyval list*⟩ should consist of one or more key names along with an associated key *property*. The properties of a key determine how it acts. The individual properties are described in the following text; a typical use of `\keys_define:nn` might read

```
\keys_define:nn { mymodule }
  {
    keyname .code:n = Some~code~using~#1,
    keyname .value_required:n = true
  }
```

where the properties of the key begin from the `.` after the key name.

The various properties available take either no arguments at all, or require one or more arguments. This is indicated in the name of the property using an argument specification. In the following discussion, each property is illustrated attached to an arbitrary ⟨*key*⟩, which when used may be supplied with a ⟨*value*⟩. All key *definitions* are local.

Key properties are applied in the reading order and so the ordering is significant. Key properties which define "actions", such as `.code:n`, `.tl_set:N`, *etc.*, override one another. Some other properties are mutually exclusive, notably `.value_required:n` and `.value_forbidden:n`, and so they replace one another. However, properties covering non-exclusive behaviours may be given in any order. Thus for example the following definitions are equivalent.

```
\keys_define:nn { mymodule }
  {
    keyname .code:n           = Some~code~using~#1,
    keyname .value_required:n = true
  }
\keys_define:nn { mymodule }
  {
    keyname .value_required:n = true,
    keyname .code:n           = Some~code~using~#1
  }
```

Note that all key properties define the key within the current TeX group, with an exception that the special `.undefine:` property *undefines* the key within the current TeX group.

`.bool_set:N`
`.bool_set:c`
`.bool_gset:N`
`.bool_gset:c`

Updated: 2013-07-08

⟨*key*⟩ `.bool_set:N` = ⟨*boolean variable*⟩

Defines ⟨*key*⟩ to set ⟨*boolean variable*⟩ to ⟨*value*⟩ (which must be either "`true`" or "`false`"). If the variable does not exist, it will be created globally at the point that the key is set up.

`.bool_set_inverse:N`
`.bool_set_inverse:c`
`.bool_gset_inverse:N`
`.bool_gset_inverse:c`

New: 2011-08-28
Updated: 2013-07-08

⟨*key*⟩ `.bool_set_inverse:N` = ⟨*boolean variable*⟩

Defines ⟨*key*⟩ to set ⟨*boolean variable*⟩ to the logical inverse of ⟨*value*⟩ (which must be either "`true`" or "`false`"). If the ⟨*boolean variable*⟩ does not exist, it will be created globally at the point that the key is set up.

`.choice:`

⟨*key*⟩ `.choice:`

Sets ⟨*key*⟩ to act as a choice key. Each valid choice for ⟨*key*⟩ must then be created, as discussed in section 27.3.

`.choices:nn`
`.choices:(Vn|en|on)`

New: 2011-08-21
Updated: 2013-07-10

⟨*key*⟩ `.choices:nn` = {⟨*choices*⟩} {⟨*code*⟩}

Sets ⟨*key*⟩ to act as a choice key, and defines a series ⟨*choices*⟩ which are implemented using the ⟨*code*⟩. Inside ⟨*code*⟩, `\l_keys_choice_tl` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of ⟨*choices*⟩ (indexed from 1). Choices are discussed in detail in section 27.3.

`.clist_set:N`
`.clist_set:c`
`.clist_gset:N`
`.clist_gset:c`

New: 2011-09-11

⟨*key*⟩ `.clist_set:N` = ⟨*comma list variable*⟩

Defines ⟨*key*⟩ to set ⟨*comma list variable*⟩ to ⟨*value*⟩. Spaces around commas and empty items will be stripped. If the variable does not exist, it is created globally at the point that the key is set up.

.code:n

Updated: 2013-07-10

⟨*key*⟩ .code:n = {⟨*code*⟩}

Stores the ⟨*code*⟩ for execution when ⟨*key*⟩ is used. The ⟨*code*⟩ can include one parameter (#1), which will be the ⟨*value*⟩ given for the ⟨*key*⟩.

.cs_set:Np
.cs_set:cp
.cs_set_protected:Np
.cs_set_protected:cp
.cs_gset:Np
.cs_gset:cp
.cs_gset_protected:Np
.cs_gset_protected:cp

New: 2020-01-11

⟨*key*⟩ .cs_set:Np = ⟨*control sequence*⟩ ⟨*arg. spec.*⟩

Defines ⟨*key*⟩ to set ⟨*control sequence*⟩ to have ⟨*arg. spec.*⟩ and replacement text ⟨*value*⟩.

.default:n
.default:(V|e|o)

Updated: 2013-07-09

⟨*key*⟩ .default:n = {⟨*default*⟩}

Creates a ⟨*default*⟩ value for ⟨*key*⟩, which is used if no value is given. This will be used if only the key name is given, but not if a blank ⟨*value*⟩ is given:

```
\keys_define:nn { mymodule }
  {
    key .code:n    = Hello~#1,
    key .default:n = World
  }
\keys_set:nn { mymodule }
  {
    key = Fred, % Prints 'Hello Fred'
    key,        % Prints 'Hello World'
    key = ,     % Prints 'Hello '
  }
```

The default does not affect keys where values are required or forbidden. Thus a required value cannot be supplied by a default value, and giving a default value for a key which cannot take a value does not trigger an error.

When no value is given for a key as part of \keys_set:nn, the .default:n value provides the value before key properties are considered. The only exception is when the .value_required:n property is active: a required value cannot be supplied by the default, and must be explicitly given as part of \keys_set:nn.

.dim_set:N
.dim_set:c
.dim_gset:N
.dim_gset:c

Updated: 2020-01-17

⟨*key*⟩ .dim_set:N = ⟨*dimension*⟩

Defines ⟨*key*⟩ to set ⟨*dimension*⟩ to ⟨*value*⟩ (which must a dimension expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

.fp_set:N
.fp_set:c
.fp_gset:N
.fp_gset:c

Updated: 2020-01-17

⟨*key*⟩ .fp_set:N = ⟨*floating point*⟩

Defines ⟨*key*⟩ to set ⟨*floating point*⟩ to ⟨*value*⟩ (which must a floating point expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

.groups:n

New: 2013-07-14

⟨*key*⟩ .groups:n = {⟨*groups*⟩}

Defines ⟨*key*⟩ as belonging to the ⟨*groups*⟩ (a comma-separated list). Groups provide a "secondary axis" for selectively setting keys, and are described in Section 27.7.

TEXhackers note: The ⟨*groups*⟩ argument is turned into a string then interpreted as a comma-separated list, so group names cannot contain commas nor start or end with a space character.

.inherit:n

New: 2016-11-22

⟨*key*⟩ .inherit:n = {⟨*parents*⟩}

Specifies that the ⟨*key*⟩ path should inherit the keys listed as any of the ⟨*parents*⟩ (a comma list), which can be a module or a sub-division thereof. For example, after setting

```
\keys_define:nn { foo } { test .code:n = \tl_show:n {#1} }
\keys_define:nn { } { bar .inherit:n = foo }
```

setting

```
\keys_set:nn { bar } { test = a }
```

will be equivalent to

```
\keys_set:nn { foo } { test = a }
```

Inheritance applies at point of use, not at definition, thus keys may be added to the ⟨*parent*⟩ after the use of `.inherit:n` and will be active. If more than one ⟨*parent*⟩ is specified, the presence of the ⟨*key*⟩ will be tested for each in turn, with the first successful hit taking priority.

.initial:n
.initial:(V|e|o)

Updated: 2013-07-09

⟨*key*⟩ .initial:n = {⟨*value*⟩}

Initialises the ⟨*key*⟩ with the ⟨*value*⟩, equivalent to

\keys_set:nn {⟨*module*⟩} { ⟨*key*⟩ = ⟨*value*⟩ }

.int_set:N
.int_set:c
.int_gset:N
.int_gset:c

Updated: 2020-01-17

⟨*key*⟩ .int_set:N = ⟨*integer*⟩

Defines ⟨*key*⟩ to set ⟨*integer*⟩ to ⟨*value*⟩ (which must be an integer expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

.legacy_if_set:n
.legacy_if_gset:n
.legacy_if_set_inverse:n
.legacy_if_gset_inverse:n

Updated: 2022-01-15

⟨*key*⟩ .legacy_if_set:n = ⟨*switch*⟩

Defines ⟨*key*⟩ to set legacy `\if`⟨*switch*⟩ to ⟨*value*⟩ (which must be either "`true`" or "`false`"). The ⟨*switch*⟩ is the name of the switch *without the leading* `if`.

The `inverse` versions will set the ⟨*switch*⟩ to the logical opposite of the ⟨*value*⟩.

.meta:n

Updated: 2013-07-10

⟨*key*⟩ .meta:n = {⟨*keyval list*⟩}

Makes ⟨*key*⟩ a meta-key, which will set ⟨*keyval list*⟩ in one go. The ⟨*keyval list*⟩ can refer as `#1` to the value given at the time the ⟨*key*⟩ is used (or, if no value is given, the ⟨*key*⟩'s default value).

`.meta:nn` ⟨*key*⟩ `.meta:nn = {`⟨*path*⟩`} {`⟨*keyval list*⟩`}`

New: 2013-07-10

Makes ⟨*key*⟩ a meta-key, which will set ⟨*keyval list*⟩ in one go using the ⟨*path*⟩ in place of the current one. The ⟨*keyval list*⟩ can refer as `#1` to the value given at the time the ⟨*key*⟩ is used (or, if no value is given, the ⟨*key*⟩'s default value).

`.multichoice:` ⟨*key*⟩ `.multichoice:`

New: 2011-08-21

Sets ⟨*key*⟩ to act as a multiple choice key. Each valid choice for ⟨*key*⟩ must then be created, as discussed in section 27.3.

`.multichoices:nn`
`.multichoices:(Vn|en|on)`

⟨*key*⟩ `.multichoices:nn {`⟨*choices*⟩`} {`⟨*code*⟩`}`

New: 2011-08-21
Updated: 2013-07-10

Sets ⟨*key*⟩ to act as a multiple choice key, and defines a series ⟨*choices*⟩ which are implemented using the ⟨*code*⟩. Inside ⟨*code*⟩, `\l_keys_choice_tl` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of ⟨*choices*⟩ (indexed from 1). Choices are discussed in detail in section 27.3.

`.muskip_set:N`
`.muskip_set:c`
`.muskip_gset:N`
`.muskip_gset:c`

⟨*key*⟩ `.muskip_set:N =` ⟨*muskip*⟩

New: 2019-05-05
Updated: 2020-01-17

Defines ⟨*key*⟩ to set ⟨*muskip*⟩ to ⟨*value*⟩ (which must be a muskip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

`.prop_put:N`
`.prop_put:c`
`.prop_gput:N`
`.prop_gput:c`

⟨*key*⟩ `.prop_put:N =` ⟨*property list*⟩

New: 2019-01-31

Defines ⟨*key*⟩ to put the ⟨*value*⟩ onto the ⟨*property list*⟩ stored under the ⟨*key*⟩. If the variable does not exist, it is created globally at the point that the key is set up.

`.skip_set:N`
`.skip_set:c`
`.skip_gset:N`
`.skip_gset:c`

⟨*key*⟩ `.skip_set:N =` ⟨*skip*⟩

Updated: 2020-01-17

Defines ⟨*key*⟩ to set ⟨*skip*⟩ to ⟨*value*⟩ (which must be a skip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

`.str_set:N`
`.str_set:c`
`.str_gset:N`
`.str_gset:c`

⟨*key*⟩ `.str_set:N =` ⟨*string variable*⟩

New: 2021-10-30

Defines ⟨*key*⟩ to set ⟨*string variable*⟩ to ⟨*value*⟩. If the variable does not exist, it is created globally at the point that the key is set up.

`.str_set_e:N`
`.str_set_e:c`
`.str_gset_e:N`
`.str_gset_e:c`

⟨*key*⟩ `.str_set_e:N =` ⟨*string variable*⟩

New: 2023-09-18

Defines ⟨*key*⟩ to set ⟨*string variable*⟩ to ⟨*value*⟩, which will be subjected to an `e`-type expansion (*i.e.* using `\str_set:Ne`). If the variable does not exist, it is created globally at the point that the key is set up.

.tl_set:N
.tl_set:c
.tl_gset:N
.tl_gset:c

⟨*key*⟩ .tl_set:N = ⟨*tl var*⟩

Defines ⟨*key*⟩ to set ⟨*tl var*⟩ to ⟨*value*⟩. If the variable does not exist, it is created globally at the point that the key is set up.

.tl_set_e:N
.tl_set_e:c
.tl_gset_e:N
.tl_gset_e:c

New: 2023-09-18

⟨*key*⟩ .tl_set_e:N = ⟨*tl var*⟩

Defines ⟨*key*⟩ to set ⟨*tl var*⟩ to ⟨*value*⟩, which will be subjected to an e-type expansion (*i.e.* using \tl_set:Ne). If the variable does not exist, it is created globally at the point that the key is set up.

.undefine:

New: 2015-07-14

⟨*key*⟩ .undefine:

Removes the definition of the ⟨*key*⟩ within the current TeX group.

.value_forbidden:n

New: 2015-07-14

⟨*key*⟩ .value_forbidden:n = true|false

Specifies that ⟨*key*⟩ cannot receive a ⟨*value*⟩ when used. If a ⟨*value*⟩ is given then an error will be issued. Setting the property "false" cancels the restriction.

.value_required:n

New: 2015-07-14

⟨*key*⟩ .value_required:n = true|false

Specifies that ⟨*key*⟩ must receive a ⟨*value*⟩ when used. If a ⟨*value*⟩ is not given then an error will be issued. Setting the property "false" cancels the restriction.

27.2 Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several subsets for a given module. This can be achieved either by adding a sub-division to the module name:

```
\keys_define:nn { mymodule / subset }
  { key .code:n = code }
```

or to the key name:

```
\keys_define:nn { mymodule }
  { subset / key .code:n = code }
```

As illustrated, the best choice of token for sub-dividing keys in this way is /. This is because of the method that is used to represent keys internally. Both of the above code fragments set the same key, which has full name mymodule/subset/key.

As illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

27.3 Choice and multiple choice keys

The l3keys system supports two types of choice key, in which a series of pre-defined input values are linked to varying implementations. Choice keys are usually created so that the various values are mutually-exclusive: only one can apply at any one time. "Multiple" choice keys are also supported: these allow a selection of values to be chosen at the same time.

Mutually-exclusive choices are created by setting the `.choice:` property:

```
\keys_define:nn { mymodule }
  { key .choice: }
```

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependent only on the name of the choice or the position of the choice in the list of all possibilities. Here, the keys can share the same code, and can be rapidly created using the `.choices:nn` property.

```
\keys_define:nn { mymodule }
  {
    key .choices:nn =
      { choice-a, choice-b, choice-c }
      {
        You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
        which~is~in~position~\int_use:N \l_keys_choice_int \c_space_tl
        in~the~list.
      }
  }
```

The index `\l_keys_choice_int` in the list of choices starts at 1.

`\l_keys_choice_int`
`\l_keys_choice_tl`

Inside the code block for a choice generated using `.choices:nn`, the variables `\l_keys_choice_tl` and `\l_keys_choice_int` are available to indicate the name of the current choice, and its position in the comma list. The position is indexed from 1. Note that, as with standard key code generated using `.code:n`, the value passed to the key (i.e. the choice name) is also available as `#1`.

On the other hand, it is sometimes useful to create choices which use entirely different code from one another. This can be achieved by setting the `.choice:` property of a key, then manually defining sub-keys.

```
\keys_define:nn { mymodule }
  {
    key .choice:,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
  }
```

It is possible to mix the two methods, but manually-created choices should *not* use `\l_keys_choice_tl` or `\l_keys_choice_int`. These variables do not have defined behaviour when used outside of code created using `.choices:nn` (*i.e.* anything might happen).

It is possible to allow choice keys to take values which have not previously been defined by adding code for the special `unknown` choice. The general behavior of the `unknown` key is described in Section 27.6. A typical example in the case of a choice would be to issue a custom error message:

```
\keys_define:nn { mymodule }
  {
    key .choice:,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
    key / unknown  .code:n =
      \msg_error:nneee { mymodule } { unknown-choice }
        { key }                               % Name of choice key
        { choice-a , choice-b ,  choice-c }  % Valid choices
        { \exp_not:n {#1} }                   % Invalid choice given
  }
```

Multiple choices are created in a very similar manner to mutually-exclusive choices, using the properties `.multichoice:` and `.multichoices:nn`. As with mutually exclusive choices, multiple choices are defined as sub-keys. Thus both

```
\keys_define:nn { mymodule }
  {
    key .multichoices:nn =
      { choice-a, choice-b, choice-c }
      {
        You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
        which~is~in~position~
        \int_use:N \l_keys_choice_int \c_space_tl
        in~the~list.
      }
  }
```

and

```
\keys_define:nn { mymodule }
  {
    key .multichoice:,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
  }
```

are valid.

When a multiple choice key is set

```
\keys_set:nn { mymodule }
  {
    key = { a , b , c } % 'key' defined as a multiple choice
  }
```

each choice is applied in turn, equivalent to a `clist` mapping or to applying each value individually:

```
\keys_set:nn { mymodule }
  {
    key = a ,
    key = b ,
    key = c ,
  }
```

Thus each separate choice will have passed to it the `\l_keys_choice_tl` and `\l_keys_-choice_int` in exactly the same way as described for `.choices:nn`.

27.4 Key usage scope

Some keys will be used as settings which have a strictly limited scope of usage. Some will be only available once, others will only be valid until typesetting begins. To allow formats to support this in a structured way, l3keys allows this information to be specified using the `.usage:n` property.

`.usage:n`

New: 2022-01-10

⟨*key*⟩ `.usage:n =` ⟨*scope*⟩

Defines the ⟨*key*⟩ to have usage within the ⟨*scope*⟩, which should be one of `general`, `preamble` or `load`.

`\l_keys_usage_load_prop`
`\l_keys_usage_preamble_prop`

New: 2022-01-10

l3keys itself does *not* attempt to redefine keys based on the usage scope. Rather, this information is made available with these two property lists. These hold an entry for each module (prefix); the value of each entry is a comma-separated list of the usage-restricted key(s).

27.5 Setting keys

`\keys_set:nn`
`\keys_set:(nV|nv|ne|no)`

Updated: 2017-11-14

`\keys_set:nn {`⟨*module*⟩`} {`⟨*keyval list*⟩`}`

Parses the ⟨*keyval list*⟩, and sets those keys which are defined for ⟨*module*⟩. The behaviour on finding an unknown key can be set by defining a special `unknown` key: this is illustrated later.

\l_keys_path_str
\l_keys_key_str
\l_keys_value_tl

Updated: 2020-02-08

For each key processed, information of the full *path* of the key, the *name* of the key and the *value* of the key is available within two string and one token list variables. These may be used within the code of the key.

The *path* of the key is a "full" description of the key, and is unique for each key. It consists of the module and full key name, thus for example

```
\keys_set:nn { mymodule } { key-a = some-value }
```

has path `mymodule/key-a` while

```
\keys_set:nn { mymodule } { subset  / key-a = some-value }
```

has path `mymodule/subset/key-a`. This information is stored in `\l_keys_path_str`.

The *name* of the key is the part of the path after the last `/`, and thus is not unique. In the preceding examples, both keys have name `key-a` despite having different paths. This information is stored in `\l_keys_key_str`.

The *value* is everything after the `=`, which may be empty if no value was given. This is stored in `\l_keys_value_tl`, and is not processed in any way by `\keys_set:nn`.

27.6 Handling of unknown keys

If a key has not previously been defined (is unknown), `\keys_set:nn` looks for a special `unknown` key for the same module, and if this is not defined raises an error indicating that the key name was unknown. This mechanism can be used for example to issue custom error texts. The `unknown` key also supports the `.default:n` property.

```
\keys_define:nn { mymodule }
  {
    unknown .code:n =
      You~tried~to~set~key~'\l_keys_key_str'~to~'#1'. ,
    unknown .default:V = \c_novalue_tl
  }
```

27.7 Selective key setting

In some cases it may be useful to be able to select only some keys for setting, even though these keys have the same path. For example, with a set of keys defined using

```
\keys_define:nn { mymodule }
  {
    key-one   .code:n   = { \my_func:n {#1} } ,
    key-two   .tl_set:N = \l_my_a_tl          ,
    key-three .tl_set:N = \l_my_b_tl          ,
    key-four  .fp_set:N = \l_my_a_fp          ,
  }
```

the use of `\keys_set:nn` attempts to set all four keys. However, in some contexts it may only be sensible to set some keys, or to control the order of setting. To do this, keys may be assigned to *groups*: arbitrary sets which are independent of the key tree. Thus modifying the example to read

```
\keys_define:nn { mymodule }
  {
    key-one   .code:n   = { \my_func:n {#1} } ,
    key-one   .groups:n = { first }           ,
    key-two   .tl_set:N = \l_my_a_tl          ,
    key-two   .groups:n = { first }           ,
    key-three .tl_set:N = \l_my_b_tl          ,
    key-three .groups:n = { second }          ,
    key-four  .fp_set:N = \l_my_a_fp          ,
  }
```

assigns `key-one` and `key-two` to group `first`, `key-three` to group `second`, while `key-four` is not assigned to a group.

Selective key setting may be achieved either by selecting one or more groups to be made "active", or by marking one or more groups to be ignored in key setting.

`\keys_set_known:nn`
`\keys_set_known:(nV|nv|ne|no)`
`\keys_set_known:nnN`
`\keys_set_known:(nVN|nvN|neN|noN)`
`\keys_set_known:nnnN`
`\keys_set_known:(nVnN|nvnN|nenN|nonN)`

New: 2011-08-23
Updated: 2019-01-29

`\keys_set_known:nn {⟨module⟩} {⟨keyval list⟩}`
`\keys_set_known:nnN {⟨module⟩} {⟨keyval list⟩} ⟨tl var⟩`
`\keys_set_known:nnnN {⟨module⟩} {⟨keyval list⟩} {⟨root⟩} ⟨tl var⟩`

These functions set keys which are known for the ⟨*module*⟩, and simply ignore other keys. The `\keys_set_known:nn` function parses the ⟨*keyval list*⟩, and sets those keys which are defined for ⟨*module*⟩. Any keys which are unknown are not processed further by the parser.

In addition, `\keys_set_known:nnN` and `\keys_set_known:nnnN` store the key–value pairs for unknown keys in the ⟨*tl var*⟩ in comma-separated form (*i.e.* an edited version of the ⟨*keyval list*⟩). When a ⟨*root*⟩ is given (`\keys_set_known:nnnN`), the key–value entries are returned relative to this point in the key tree. When it is absent, only the key name and value are provided. The correct list is returned by nested calls.

\keys_set_groups:nnn
\keys_set_groups:(nnV|nnv|nno)
\keys_set_groups:nnnN
\keys_set_groups:(nnVN|nnvN|nnoN)
\keys_set_groups:nnnnN
\keys_set_groups:(nnVnN|nnvnN|nnonN)

New: 2013-07-14
Updated: 2024-05-08

\keys_set_groups:nnn {⟨*module*⟩} {⟨*groups*⟩} {⟨*keyval list*⟩}
\keys_set_groups:nnnN {⟨*module*⟩} {⟨*groups*⟩} {⟨*keyval list*⟩} ⟨*tl var*⟩
\keys_set_groups:nnnnN {⟨*module*⟩} {⟨*groups*⟩} {⟨*keyval list*⟩} {⟨*root*⟩} ⟨*tl var*⟩

These functions activate key selection in an "opt-in" sense: only keys assigned to one or more of the ⟨*groups*⟩ specified are set. The ⟨*groups*⟩ are given as a comma-separated list. Unknown keys are not assigned to any group and are thus never set.

In addition, \keys_set_groups:nnnN and \keys_set_groups:nnnnN store the key–value pairs for skipped keys in the ⟨*tl var*⟩ in comma-separated form (*i.e.* an edited version of the ⟨*keyval list*⟩). When a ⟨*root*⟩ is given (\keys_set_groups:nnnnN), the key–value entries are returned relative to this point in the key tree. When it is absent, only the key name and value are provided. The correct list is returned by nested calls.

\keys_set_exclude_groups:nnn
\keys_set_exclude_groups:(nnV|nnv|nno)
\keys_set_exclude_groups:nnnN
\keys_set_exclude_groups:(nnVN|nnvN|nnoN)
\keys_set_exclude_groups:nnnnN
\keys_set_exclude_groups:(nnVnN|nnvnN|nnonN)

New: 2024-01-10

\keys_set_exclude_groups:nnn {⟨*module*⟩} {⟨*groups*⟩} {⟨*keyval list*⟩}
\keys_set_exclude_groups:nnnN {⟨*module*⟩} {⟨*groups*⟩} {⟨*keyval list*⟩} ⟨*tl var*⟩
\keys_set_exclude_groups:nnnnN {⟨*module*⟩} {⟨*groups*⟩} {⟨*keyval list*⟩} {⟨*root*⟩} ⟨*tl var*⟩

These functions activate key selection in an "opt-out" sense: keys assigned to one or more of the ⟨*groups*⟩ specified are *not* set. The ⟨*groups*⟩ are given as a comma-separated list. Unknown keys are not assigned to any group and are thus always set.

In addition, \keys_set_exclude_groups:nnnN and \keys_set_exclude_groups:nnnnN store the key–value pairs for skipped keys in the ⟨*tl var*⟩ in comma-separated form (*i.e.* an edited version of the ⟨*keyval list*⟩). When a ⟨*root*⟩ is given (\keys_set_exclude_groups:nnnnN), the key–value entries are returned relative to this point in the key tree. When it is absent, only the key name and value are provided. The correct list is returned by nested calls.

27.8 Digesting keys

\keys_precompile:nnN

New: 2022-03-09

\keys_precompile:nnN {⟨*module*⟩} {⟨*keyval list*⟩} ⟨*tl var*⟩

Parses the ⟨*keyval list*⟩ as for \keys_set:nn, placing the resulting code for those which set variables or functions into the ⟨*tl var*⟩. Thus this function "precompiles" the keyval list into a set of results which can be applied rapidly.

27.9 Utility functions for keys

\keys_if_exist_p:nn ⋆
\keys_if_exist_p:ne ⋆
\keys_if_exist:nnTF ⋆
\keys_if_exist:neTF ⋆

Updated: 2022-01-10

\keys_if_exist_p:nn {⟨*module*⟩} {⟨*key*⟩}
\keys_if_exist:nnTF {⟨*module*⟩} {⟨*key*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*key*⟩ exists for ⟨*module*⟩, *i.e.* if any code has been defined for ⟨*key*⟩.

\keys_if_choice_exist_p:nnn ⋆
\keys_if_choice_exist:nnnTF ⋆

New: 2011-08-21
Updated: 2017-11-14

\keys_if_choice_exist_p:nnn {⟨*module*⟩} {⟨*key*⟩} {⟨*choice*⟩}
\keys_if_choice_exist:nnnTF {⟨*module*⟩} {⟨*key*⟩} {⟨*choice*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*choice*⟩ is defined for the ⟨*key*⟩ within the ⟨*module*⟩, *i.e.* if any code has been defined for ⟨*key*⟩/⟨*choice*⟩. The test is `false` if the ⟨*key*⟩ itself is not defined.

\keys_show:nn

Updated: 2015-08-09

\keys_show:nn {⟨*module*⟩} {⟨*key*⟩}

Displays in the terminal the information associated to the ⟨*key*⟩ for a ⟨*module*⟩, including the function which is used to actually implement it.

\keys_log:nn

New: 2014-08-22
Updated: 2015-08-09

\keys_log:nn {⟨*module*⟩} {⟨*key*⟩}

Writes in the log file the information associated to the ⟨*key*⟩ for a ⟨*module*⟩. See also `\keys_show:nn` which displays the result in the terminal.

27.10 Low-level interface for parsing key–val lists

To re-cap from earlier, a key–value list is input of the form

```
KeyOne = ValueOne ,
KeyTwo = ValueTwo ,
KeyThree
```

where each key–value pair is separated by a comma from the rest of the list, and each key–value pair does not necessarily contain an equals sign or a value! Processing this type of input correctly requires a number of careful steps, to correctly account for braces, spaces and the category codes of separators.

While the functions described earlier are used as a high-level interface for processing such input, in special circumstances you may wish to use a lower-level approach. The low-level parsing system converts a ⟨*key-value list*⟩ into ⟨*keys*⟩ and associated ⟨*values*⟩. After the parsing phase is completed, the resulting keys and values (or keys alone) are available for further processing. This processing is not carried out by the low-level parser itself, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (it receives two arguments), and a second function is required for keys given without any value (it is called with a single argument).

The parser does not double `#` tokens or expand any input. Active tokens `=` and `,` appearing at the outer level of braces are converted to category "other" (12) so that the parser does not "miss" any due to category code changes. Spaces are removed from the ends of the keys and values. Keys and values which are given in braces have exactly one set removed (after space trimming), thus

```
key = {value here},
```

and

```
key = value here,
```

are treated identically.

`\keyval_parse:nnn` ☆
`\keyval_parse:(nnV|nnv)` ☆

New: 2020-12-19
Updated: 2021-05-10

`\keyval_parse:nnn {⟨code₁⟩} {⟨code₂⟩} {⟨key-value list⟩}`

Parses the ⟨*key-value list*⟩ into a series of ⟨*keys*⟩ and associated ⟨*values*⟩, or keys alone (if no ⟨*value*⟩ was given). ⟨*code*$_1$⟩ receives each ⟨*key*⟩ (with no ⟨*value*⟩) as a trailing brace group, whereas ⟨*code*$_2$⟩ is appended by two brace groups, the ⟨*key*⟩ and ⟨*value*⟩. The order of the ⟨*keys*⟩ in the ⟨*key-value list*⟩ is preserved. Thus

```
\keyval_parse:nnn
  { \use_none:nn  { code 1 } }
  { \use_none:nnn { code 2 } }
  { key1 = value1 , key2 = value2, key3 = , key4 }
```

is converted into an input stream

```
\use_none:nnn { code 2 } { key1 } { value1 }
\use_none:nnn { code 2 } { key2 } { value2 }
\use_none:nnn { code 2 } { key3 } { }
\use_none:nn  { code 1 } { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the ⟨*key*⟩ and ⟨*value*⟩, then one *outer* set of braces is removed from the ⟨*key*⟩ and ⟨*value*⟩ as part of the processing. If you need exactly the output shown above, you'll need to either `e`-type or `x`-type expand the function.

TEXhackers note: The result of each list element is returned within `\exp_not:n`, which means that the converted input stream does not expand further when appearing in an `e`-type or `x`-type argument expansion.

`\keyval_parse:NNn` ☆
`\keyval_parse:(NNV|NNv)` ☆

Updated: 2021-05-10

`\keyval_parse:NNn` ⟨*function*$_1$⟩ ⟨*function*$_2$⟩ {⟨*key-value list*⟩}

Parses the ⟨*key-value list*⟩ into a series of ⟨*keys*⟩ and associated ⟨*values*⟩, or keys alone (if no ⟨*value*⟩ was given). ⟨*function*$_1$⟩ should take one argument, while ⟨*function*$_2$⟩ should absorb two arguments. After `\keyval_parse:NNn` has parsed the ⟨*key-value list*⟩, ⟨*function*$_1$⟩ is used to process keys given with no value and ⟨*function*$_2$⟩ is used to process keys given with a value. The order of the ⟨*keys*⟩ in the ⟨*key-value list*⟩ is preserved. Thus

```
\keyval_parse:NNn \function:n \function:nn
  { key1 = value1 , key2 = value2, key3 = , key4 }
```

is converted into an input stream

```
\function:nn { key1 } { value1 }
\function:nn { key2 } { value2 }
\function:nn { key3 } { }
\function:n  { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the ⟨*key*⟩ and ⟨*value*⟩, then one *outer* set of braces is removed from the ⟨*key*⟩ and ⟨*value*⟩ as part of the processing.

This shares the implementation of `\keyval_parse:nnn`, the difference is only semantically.

TEXhackers note: The result is returned within `\exp_not:n`, which means that the converted input stream does not expand further when appearing in an `e`-type or `x`-type argument expansion.

Chapter 28

The l3intarray module Fast global integer arrays

For applications requiring heavy use of integers, this module provides arrays which can be accessed in constant time (contrast l3seq, where access time is linear). These arrays have several important features

- The size of the array is fixed and must be given at point of initialisation
- The absolute value of each entry has maximum $2^{30} - 1$ (*i.e.* one power lower than the usual `\c_max_int` ceiling of $2^{31} - 1$)

The use of `intarray` data is therefore recommended for cases where the need for fast access is of paramount importance.

28.1 Creating and initialising integer array variables

`\intarray_new:Nn`
`\intarray_new:cn`
New: 2018-03-29

`\intarray_new:Nn ⟨intarray var⟩ {⟨size⟩}`

Evaluates the integer expression ⟨*size*⟩ and allocates an ⟨*integer array variable*⟩ with that number of (zero) entries. The variable name should start with `\g_` because assignments are always global.

`\intarray_const_from_clist:Nn`
`\intarray_const_from_clist:cn`
New: 2018-05-04

`\intarray_const_from_clist:Nn ⟨intarray var⟩ {⟨int expr clist⟩}`

Creates a new constant ⟨*integer array variable*⟩ or raises an error if the name is already taken. The ⟨*integer array variable*⟩ is set (globally) to contain as its items the results of evaluating each ⟨*integer expression*⟩ in the ⟨*comma list*⟩.

`\intarray_gzero:N`
`\intarray_gzero:c`
New: 2018-05-04

`\intarray_gzero:N ⟨intarray var⟩`

Sets all entries of the ⟨*integer array variable*⟩ to zero. Assignments are always global.

28.2 Adding data to integer arrays

`\intarray_gset:Nnn`
`\intarray_gset:cnn`
New: 2018-03-29

`\intarray_gset:Nnn` ⟨*intarray var*⟩ {⟨*position*⟩} {⟨*value*⟩}

Stores the result of evaluating the integer expression ⟨*value*⟩ into the ⟨*integer array variable*⟩ at the (integer expression) ⟨*position*⟩. If the ⟨*position*⟩ is not between 1 and the `\intarray_count:N`, or the ⟨*value*⟩'s absolute value is bigger than $2^{30} - 1$, an error occurs. Assignments are always global.

28.3 Counting entries in integer arrays

`\intarray_count:N` ⋆
`\intarray_count:c` ⋆
New: 2018-03-29

`\intarray_count:N` ⟨*intarray var*⟩

Expands to the number of entries in the ⟨*integer array variable*⟩. Contrarily to `\seq_count:N` this is performed in constant time.

28.4 Using a single entry

`\intarray_item:Nn` ⋆
`\intarray_item:cn` ⋆
New: 2018-03-29

`\intarray_item:Nn` ⟨*intarray var*⟩ {⟨*position*⟩}

Expands to the integer entry stored at the (integer expression) ⟨*position*⟩ in the ⟨*integer array variable*⟩. If the ⟨*position*⟩ is not between 1 and the `\intarray_count:N`, an error occurs.

`\intarray_rand_item:N` ⋆
`\intarray_rand_item:c` ⋆
New: 2018-05-05

`\intarray_rand_item:N` ⟨*intarray var*⟩

Selects a pseudo-random item of the ⟨*integer array*⟩. If the ⟨*integer array*⟩ is empty, produce an error.

28.5 Integer array conditional

`\intarray_if_exist_p:N` ⋆
`\intarray_if_exist_p:c` ⋆
`\intarray_if_exist:NTF` ⋆
`\intarray_if_exist:cTF` ⋆
New: 2024-03-31

`\intarray_if_exist_p:N` ⟨*intarray var*⟩
`\intarray_if_exist:NTF` ⟨*intarray var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*intarray var*⟩ is currently defined. This does not check that the ⟨*intarray var*⟩ really is an integer array variable.

28.6 Viewing integer arrays

`\intarray_show:N`
`\intarray_show:c`
`\intarray_log:N`
`\intarray_log:c`
New: 2018-05-04

`\intarray_show:N` ⟨*intarray var*⟩
`\intarray_log:N` ⟨*intarray var*⟩

Displays the items in the ⟨*integer array variable*⟩ in the terminal or writes them in the log file.

28.7 Implementation notes

It is a wrapper around the `\fontdimen` primitive, used to store arrays of integers (with a restricted range: absolute value at most $2^{30}-1$). In contrast to l3seq sequences the access to individual entries is done in constant time rather than linear time, but only integers can be stored. More precisely, the primitive `\fontdimen` stores dimensions but the l3intarray module transparently converts these from/to integers. Assignments are always global.

While LuaTEX's memory is extensible, other engines can "only" deal with a bit less than 4×10^6 entries in all `\fontdimen` arrays combined (with default TEX Live settings).

Chapter 29

The l3fp module Floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. *Floating point expressions* ("⟨`fp expr`⟩") support the following operations with their usual precedence.

- Basic arithmetic: addition $x + y$, subtraction $x - y$, multiplication $x * y$, division x/y, square root $\sqrt{x}$, and parentheses.
- Comparison operators: $x < y$, $x <= y$, $x >? y$, $x \,!= y$ *etc.*
- Boolean logic: sign $\operatorname{sign} x$, negation $!\,x$, conjunction $x \,\&\&\, y$, disjunction $x \,||\, y$, ternary operator $x \,?\, y : z$.
- Exponentials: $\exp x$, $\ln x$, x^y, $\operatorname{logb} x$.
- Integer factorial: $\operatorname{fact} x$.
- Trigonometry: $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\sec x$, $\csc x$ expecting their arguments in radians, and $\operatorname{sind} x$, $\operatorname{cosd} x$, $\operatorname{tand} x$, $\operatorname{cotd} x$, $\operatorname{secd} x$, $\operatorname{cscd} x$ expecting their arguments in degrees.
- Inverse trigonometric functions: $\operatorname{asin} x$, $\operatorname{acos} x$, $\operatorname{atan} x$, $\operatorname{acot} x$, $\operatorname{asec} x$, $\operatorname{acsc} x$ giving a result in radians, and $\operatorname{asind} x$, $\operatorname{acosd} x$, $\operatorname{atand} x$, $\operatorname{acotd} x$, $\operatorname{asecd} x$, $\operatorname{acscd} x$ giving a result in degrees.

(not yet) Hyperbolic functions and their inverse functions: $\sinh x$, $\cosh x$, $\tanh x$, $\coth x$, $\operatorname{sech} x$, csch, and $\operatorname{asinh} x$, $\operatorname{acosh} x$, $\operatorname{atanh} x$, $\operatorname{acoth} x$, $\operatorname{asech} x$, $\operatorname{acsch} x$.

- Extrema: $\max(x_1, x_2, \ldots)$, $\min(x_1, x_2, \ldots)$, $\operatorname{abs}(x)$.
- Rounding functions, controlled by two optional values, n (number of places, 0 by default) and t (behavior on a tie, `nan` by default):
  - $\operatorname{trunc}(x, n)$ rounds towards zero,
  - $\operatorname{floor}(x, n)$ rounds towards $-\infty$,

- ceil(x, n) rounds towards $+\infty$,
- round(x, n, t) rounds to the closest value, with ties rounded to an even value by default, towards zero if $t = 0$, towards $+\infty$ if $t > 0$ and towards $-\infty$ if $t < 0$.

And *(not yet)* modulo, and "quantize".

- Random numbers: rand(), randint(m, n).
- Constants: `pi`, `deg` (one degree in radians).
- Dimensions, automatically expressed in points, *e.g.*, `pc` is 12.
- Automatic conversion (no need for `\⟨type⟩_use:N`) of integer, dimension, and skip variables to floating point numbers, expressing dimensions in points and ignoring the stretch and shrink components of skips.
- Tuples: $(x_1, \ldots, x_n)$ that can be stored in variables, added together, multiplied or divided by a floating point number, and nested.

Floating point numbers can be given either explicitly (in a form such as `1.234e-34`, or `-.0001`), or as a stored floating point variable, which is automatically replaced by its current value. A "floating point" is a floating point number or a tuple thereof. See section 29.12.1 for a description of what a floating point is, section 29.12.2 for details about how an expression is parsed, and section 29.12.3 to know what the various operations do. Some operations may raise exceptions (error messages), described in section 29.10.

An example of use could be the following.

```
\LaTeX{} can now compute: $ \frac{\sin (3.5)}{2} + 2\cdot 10^{-3}
= \ExplSyntaxOn \fp_to_decimal:n {sin(3.5)/2 + 2e-3} $.
```

The operation `round` can be used to limit the result's precision. Adding $+0$ avoids the possibly undesirable output `-0`, replacing it by `+0`. However, the l3fp module is mostly meant as an underlying tool for higher-level commands. For example, one could provide a function to typeset nicely the result of floating point computations.

```
\documentclass{article}
\usepackage{siunitx}
\ExplSyntaxOn
\NewDocumentCommand { \calcnum } { m }
  { \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\begin{document}
\calcnum { 2 pi * sin ( 2.3 ^ 5 ) }
\end{document}
```

See the documentation of siunitx for various options of `\num`.

29.1 Creating and initialising floating point variables

\fp_new:N
\fp_new:c

Updated: 2012-05-08

\fp_new:N ⟨*fp var*⟩

Creates a new ⟨*fp var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*fp var*⟩ is initially +0.

\fp_const:Nn
\fp_const:cn

Updated: 2012-05-08

\fp_const:Nn ⟨*fp var*⟩ {⟨*fp expr*⟩}

Creates a new constant ⟨*fp var*⟩ or raises an error if the name is already taken. The ⟨*fp var*⟩ is set globally equal to the result of evaluating the ⟨*fp expr*⟩.

\fp_zero:N
\fp_zero:c
\fp_gzero:N
\fp_gzero:c

Updated: 2012-05-08

\fp_zero:N ⟨*fp var*⟩

Sets the ⟨*fp var*⟩ to +0.

\fp_zero_new:N
\fp_zero_new:c
\fp_gzero_new:N
\fp_gzero_new:c

Updated: 2012-05-08

\fp_zero_new:N ⟨*fp var*⟩

Ensures that the ⟨*fp var*⟩ exists globally by applying \fp_new:N if necessary, then applies \fp_(g)zero:N to leave the ⟨*fp var*⟩ set to +0.

29.2 Setting floating point variables

\fp_set:Nn
\fp_set:(cn|NV|cV)
\fp_gset:Nn
\fp_gset:(cn|NV|cV)

Updated: 2012-05-08

\fp_set:Nn ⟨*fp var*⟩ {⟨*fp expr*⟩}

Sets ⟨*fp var*⟩ equal to the result of computing the ⟨*fp expr*⟩.

\fp_set_eq:NN
\fp_set_eq:(cN|Nc|cc)
\fp_gset_eq:NN
\fp_gset_eq:(cN|Nc|cc)

Updated: 2012-05-08

\fp_set_eq:NN ⟨*fp var*$_1$⟩ ⟨*fp var*$_2$⟩

Sets the floating point variable ⟨*fp var*$_1$⟩ equal to the current value of ⟨*fp var*$_2$⟩.

\fp_add:Nn
\fp_add:cn
\fp_gadd:Nn
\fp_gadd:cn

Updated: 2012-05-08

\fp_add:Nn ⟨*fp var*⟩ {⟨*fp expr*⟩}

Adds the result of computing the ⟨*fp expr*⟩ to the ⟨*fp var*⟩. This also applies if ⟨*fp var*⟩ and ⟨*floating point expression*⟩ evaluate to tuples of the same size.

\fp_sub:Nn
\fp_sub:cn
\fp_gsub:Nn
\fp_gsub:cn

Updated: 2012-05-08

\fp_sub:Nn ⟨*fp var*⟩ {⟨*fp expr*⟩}

Subtracts the result of computing the ⟨*floating point expression*⟩ from the ⟨*fp var*⟩. This also applies if ⟨*fp var*⟩ and ⟨*floating point expression*⟩ evaluate to tuples of the same size.

29.3 Using floating points

\fp_eval:n ⋆

New: 2012-05-08
Updated: 2012-07-08

\fp_eval:n {⟨*fp expr*⟩}

Evaluates the ⟨*fp expr*⟩ and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and nan trigger an "invalid operation" exception. For a tuple, each item is converted using \fp_eval:n and they are combined as (⟨*fp*$_1$⟩,␣⟨*fp*$_2$⟩,␣...⟨*fp*$_n$⟩) if $n > 1$ and (⟨*fp*$_1$⟩,) or () for fewer items. This function is identical to \fp_to_decimal:n.

\fp_sign:n ⋆

New: 2018-11-03

\fp_sign:n {⟨*fp expr*⟩}

Evaluates the ⟨*fp expr*⟩ and leaves its sign in the input stream using \fp_eval:n {sign(⟨*result*⟩)}: $+1$ for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ± 0 for ± 0. If the operand is a tuple or is nan, then "invalid operation" occurs and the result is 0.

\fp_to_decimal:N ⋆
\fp_to_decimal:c ⋆
\fp_to_decimal:n ⋆

New: 2012-05-08
Updated: 2012-07-08

\fp_to_decimal:N ⟨*fp var*⟩
\fp_to_decimal:n {⟨*fp expr*⟩}

Evaluates the ⟨*fp expr*⟩ and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and nan trigger an "invalid operation" exception. For a tuple, each item is converted using \fp_to_decimal:n and they are combined as (⟨*fp*$_1$⟩,␣⟨*fp*$_2$⟩,␣...⟨*fp*$_n$⟩) if $n > 1$ and (⟨*fp*$_1$⟩,) or () for fewer items.

\fp_to_dim:N ⋆
\fp_to_dim:c ⋆
\fp_to_dim:n ⋆

Updated: 2016-03-22

\fp_to_dim:N ⟨*fp var*⟩
\fp_to_dim:n {⟨*fp expr*⟩}

Evaluates the ⟨*fp expr*⟩ and expresses the result as a dimension (in pt) suitable for use in dimension expressions. The output is identical to \fp_to_decimal:n, with an additional trailing pt (both letter tokens). In particular, the result may be outside the range $[-2^{14}+2^{-17}, 2^{14}-2^{-17}]$ of valid TeX dimensions, leading to overflow errors if used as a dimension. Tuples, as well as the values $\pm\infty$ and nan, trigger an "invalid operation" exception.

\fp_to_int:N ⋆
\fp_to_int:c ⋆
\fp_to_int:n ⋆

Updated: 2012-07-08

\fp_to_int:N ⟨*fp var*⟩
\fp_to_int:n {⟨*fp expr*⟩}

Evaluates the ⟨*fp expr*⟩, and rounds the result to the closest integer, rounding exact ties to an even integer. The result may be outside the range $[-2^{31}+1, 2^{31}-1]$ of valid TeX integers, leading to overflow errors if used in an integer expression. Tuples, as well as the values $\pm\infty$ and nan, trigger an "invalid operation" exception.

\fp_to_scientific:N ⋆
\fp_to_scientific:c ⋆
\fp_to_scientific:n ⋆

New: 2012-05-08
Updated: 2016-03-22

\fp_to_scientific:N ⟨fp var⟩
\fp_to_scientific:n {⟨fp expr⟩}

Evaluates the ⟨*fp expr*⟩ and expresses the result in scientific notation:

⟨*optional -*⟩⟨*digit*⟩.⟨*15 digits*⟩e⟨*optional sign*⟩⟨*exponent*⟩

The leading ⟨*digit*⟩ is non-zero except in the case of ± 0. The values $\pm\infty$ and `nan` trigger an "invalid operation" exception. Normal category codes apply: thus the `e` is category code 11 (a letter). For a tuple, each item is converted using `\fp_to_scientific:n` and they are combined as `(`⟨*fp*$_1$⟩`,␣`⟨*fp*$_2$⟩`,␣`...⟨*fp*$_n$⟩`)` if $n > 1$ and `(`⟨*fp*$_1$⟩`,)` or `()` for fewer items.

\fp_to_tl:N ⋆
\fp_to_tl:c ⋆
\fp_to_tl:n ⋆

Updated: 2016-03-22

\fp_to_tl:N ⟨fp var⟩
\fp_to_tl:n {⟨fp expr⟩}

Evaluates the ⟨*fp expr*⟩ and expresses the result in (almost) the shortest possible form. Numbers in the ranges $(0, 10^{-3})$ and $[10^{16}, \infty)$ are expressed in scientific notation with trailing zeros trimmed and no decimal separator when there is a single significant digit (this differs from `\fp_to_scientific:n`). Numbers in the range $[10^{-3}, 10^{16})$ are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see `\fp_to_decimal:n`. Negative numbers start with `-`. The special values ± 0, $\pm\infty$ and `nan` are rendered as `0`, `-0`, `inf`, `-inf`, and `nan` respectively. Normal category codes apply and thus `inf` or `nan`, if produced, are made up of letters. For a tuple, each item is converted using `\fp_to_tl:n` and they are combined as `(`⟨*fp*$_1$⟩`,␣`⟨*fp*$_2$⟩`,␣`...⟨*fp*$_n$⟩`)` if $n > 1$ and `(`⟨*fp*$_1$⟩`,)` or `()` for fewer items.

\fp_use:N ⋆
\fp_use:c ⋆

Updated: 2012-07-08

\fp_use:N ⟨fp var⟩

Inserts the value of the ⟨*fp var*⟩ into the input stream as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values $\pm\infty$ and `nan` trigger an "invalid operation" exception. For a tuple, each item is converted using `\fp_to_decimal:n` and they are combined as `(`⟨*fp*$_1$⟩`,␣`⟨*fp*$_2$⟩`,␣`...⟨*fp*$_n$⟩`)` if $n > 1$ and `(`⟨*fp*$_1$⟩`,)` or `()` for fewer items. This function is identical to `\fp_to_decimal:N`.

29.4 Floating point conditionals

\fp_if_exist_p:N ⋆
\fp_if_exist_p:c ⋆
\fp_if_exist:N*TF* ⋆
\fp_if_exist:c*TF* ⋆

Updated: 2012-05-08

\fp_if_exist_p:N ⟨fp var⟩
\fp_if_exist:NTF ⟨fp var⟩ {⟨true code⟩} {⟨false code⟩}

Tests whether the ⟨*fp var*⟩ is currently defined. This does not check that the ⟨*fp var*⟩ really is a floating point variable.

`\fp_compare_p:nNn` ⋆ `\fp_compare:nNnTF` ⋆

`\fp_compare_p:nNn {`⟨*fp expr*$_1$⟩`}` ⟨*relation*⟩ `{`⟨*fp expr*$_2$⟩`}`
`\fp_compare:nNnTF {`⟨*fp expr*$_1$⟩`}` ⟨*relation*⟩ `{`⟨*fp expr*$_2$⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Updated: 2012-05-08

Compares the ⟨*fp expr*$_1$⟩ and the ⟨*fp expr*$_2$⟩, and returns `true` if the ⟨*relation*⟩ is obeyed. Two floating points x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or $x?y$ ("not ordered"). The last case occurs exactly if one or both operands is `nan` or is a tuple, unless they are equal tuples. Note that a `nan` is distinct from any value, even another `nan`, hence $x = x$ is not true for a `nan`. To test if a value is `nan`, compare it to an arbitrary number with the "not ordered" relation.

```
\fp_compare:nNnTF { <value> } ? { 0 }
  { } % <value> is nan
  { } % <value> is not nan
```

Tuples are equal if they have the same number of items and items compare equal (in particular there must be no `nan`). At present any other comparison with tuples yields `?` (not ordered). This is experimental.

This function is less flexible than `\fp_compare:nTF` but slightly faster. It is provided for consistency with `\int_compare:nNnTF` and `\dim_compare:nNnTF`.

\fp_compare_p:n ⋆
\fp_compare:nTF ⋆

Updated: 2013-12-14

```
\fp_compare_p:n
  {
    ⟨fp expr_1⟩ ⟨relation_1⟩
    ...
    ⟨fp expr_N⟩ ⟨relation_N⟩
    ⟨fp expr_N+1⟩
  }
\fp_compare:nTF
  {
    ⟨fp expr_1⟩ ⟨relation_1⟩
    ...
    ⟨fp expr_N⟩ ⟨relation_N⟩
    ⟨fp expr_N+1⟩
  }
  {⟨true code⟩} {⟨false code⟩}
```

Evaluates the ⟨*fp exprs*⟩ as described for `\fp_eval:n` and compares consecutive result using the corresponding ⟨*relation*⟩, namely it compares ⟨*fp expr*$_1$⟩ and ⟨*fp expr*$_2$⟩ using the ⟨*relation*$_1$⟩, then ⟨*fp expr*$_2$⟩ and ⟨*fp expr*$_3$⟩ using the ⟨*relation*$_2$⟩, until finally comparing ⟨*fp expr*$_N$⟩ and ⟨*fp expr*$_{N+1}$⟩ using the ⟨*relation*$_N$⟩. The test yields `true` if all comparisons are `true`. Each ⟨*floating point expression*⟩ is evaluated only once. Contrarily to `\int_compare:nTF`, all ⟨*fp exprs*⟩ are computed, even if one comparison is `false`. Two floating points x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or $x?y$ ("not ordered"). The last case occurs exactly if one or both operands is `nan` or is a tuple, unless they are equal tuples. Each ⟨*relation*⟩ can be any (non-empty) combination of `<`, `=`, `>`, and `?`, plus an optional leading `!` (which negates the ⟨*relation*⟩), with the restriction that the ⟨*relation*⟩ may not start with `?`, as this symbol has a different meaning (in combination with `:`) within floating point expressions. The comparison x ⟨*relation*⟩ y is then `true` if the ⟨*relation*⟩ does not start with `!` and the actual relation (`<`, `=`, `>`, or `?`) between x and y appears within the ⟨*relation*⟩, or on the contrary if the ⟨*relation*⟩ starts with `!` and the relation between x and y does not appear within the ⟨*relation*⟩. Common choices of ⟨*relation*⟩ include `>=` (greater or equal), `!=` (not equal), `!?` or `<=>` (comparable).

This function is more flexible than `\fp_compare:nNnTF` and only slightly slower.

\fp_if_nan_p:n ⋆
\fp_if_nan:nTF ⋆

New: 2019-08-25

```
\fp_if_nan_p:n {⟨fp expr⟩}
\fp_if_nan:nTF {⟨fp expr⟩} {⟨true code⟩} {⟨false code⟩}
```

Evaluates the ⟨*fp expr*⟩ and tests whether the result is exactly `nan`. The test returns `false` for any other result, even a tuple containing `nan`.

29.5 Floating point expression loops

\fp_do_until:nNnn ☆

New: 2012-08-16

```
\fp_do_until:nNnn {⟨fp expr_1⟩} ⟨relation⟩ {⟨fp expr_2⟩} {⟨code⟩}
```

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for `\fp_compare:nNnTF`. If the test is `false` then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is `true`.

\fp_do_while:nNnn ☆ \fp_do_while:nNnn {⟨*fp expr*$_1$⟩} ⟨*relation*⟩ {⟨*fp expr*$_2$⟩} {⟨*code*⟩}

New: 2012-08-16

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for \fp_compare:nNnTF. If the test is true then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is false.

\fp_until_do:nNnn ☆ \fp_until_do:nNnn {⟨*fp expr*$_1$⟩} ⟨*relation*⟩ {⟨*fp expr*$_2$⟩} {⟨*code*⟩}

New: 2012-08-16

Evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for \fp_compare:nNnTF, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is false. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is true.

\fp_while_do:nNnn ☆ \fp_while_do:nNnn {⟨*fp expr*$_1$⟩} ⟨*relation*⟩ {⟨*fp expr*$_2$⟩} {⟨*code*⟩}

New: 2012-08-16

Evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for \fp_compare:nNnTF, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is true. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is false.

\fp_do_until:nn ☆ \fp_do_until:nn { ⟨*fp expr*$_1$⟩ ⟨*relation*⟩ ⟨*fp expr*$_2$⟩ } {⟨*code*⟩}

New: 2012-08-16
Updated: 2013-12-14

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for \fp_compare:nTF. If the test is false then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is true.

\fp_do_while:nn ☆ \fp_do_while:nn { ⟨*fp expr*$_1$⟩ ⟨*relation*⟩ ⟨*fp expr*$_2$⟩ } {⟨*code*⟩}

New: 2012-08-16
Updated: 2013-12-14

Places the ⟨*code*⟩ in the input stream for TEX to process, and then evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for \fp_compare:nTF. If the test is true then the ⟨*code*⟩ is inserted into the input stream again and a loop occurs until the ⟨*relation*⟩ is false.

\fp_until_do:nn ☆ \fp_until_do:nn { ⟨*fp expr*$_1$⟩ ⟨*relation*⟩ ⟨*fp expr*$_2$⟩ } {⟨*code*⟩}

New: 2012-08-16
Updated: 2013-12-14

Evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for \fp_compare:nTF, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is false. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is true.

\fp_while_do:nn ☆ \fp_while_do:nn { ⟨*fp expr*$_1$⟩ ⟨*relation*⟩ ⟨*fp expr*$_2$⟩ } {⟨*code*⟩}

New: 2012-08-16
Updated: 2013-12-14

Evaluates the relationship between the two ⟨*floating point expressions*⟩ as described for \fp_compare:nTF, and then places the ⟨*code*⟩ in the input stream if the ⟨*relation*⟩ is true. After the ⟨*code*⟩ has been processed by TEX the test is repeated, and a loop occurs until the test is false.

\fp_step_function:nnnN ☆
\fp_step_function:nnnc ☆

New: 2016-11-21
Updated: 2016-12-06

\fp_step_function:nnnN {⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} ⟨*function*⟩

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, each of which should be a floating point expression evaluating to a floating point number, not a tuple. The ⟨*function*⟩ is then placed in front of each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩). The ⟨*step*⟩ must be non-zero. If the ⟨*step*⟩ is positive, the loop stops when the ⟨*value*⟩ becomes larger than the ⟨*final value*⟩. If the ⟨*step*⟩ is negative, the loop stops when the ⟨*value*⟩ becomes smaller than the ⟨*final value*⟩. The ⟨*function*⟩ should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\fp_step_function:nnnN { 1.0 } { 0.1 } { 1.5 } \my_func:n
```

would print

[I saw 1.0] [I saw 1.1] [I saw 1.2] [I saw 1.3] [I saw 1.4] [I saw 1.5]

TeXhackers note: Due to rounding, it may happen that adding the ⟨*step*⟩ to the ⟨*value*⟩ does not change the ⟨*value*⟩; such cases give an error, as they would otherwise lead to an infinite loop.

\fp_step_inline:nnnn

New: 2016-11-21
Updated: 2016-12-06

\fp_step_inline:nnnn {⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} {⟨*code*⟩}

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream with #1 replaced by the current ⟨*value*⟩. Thus the ⟨*code*⟩ should define a function of one argument (#1).

\fp_step_variable:nnnNn

New: 2017-04-12

\fp_step_variable:nnnNn
{⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} ⟨*tl var*⟩ {⟨*code*⟩}

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream, with the ⟨*tl var*⟩ defined as the current ⟨*value*⟩. Thus the ⟨*code*⟩ should make use of the ⟨*tl var*⟩.

29.6 Symbolic expressions

Floating point expressions support variables: these can only be set locally, so act like standard \l_... variables.

```
\fp_new_variable:n { A }
\fp_set:Nn \l_tmpb_fp { 1 * sin(A) + 3**2 }
\fp_show:n { \l_tmpb_fp }
\fp_show:N \l_tmpb_fp
\fp_set_variable:nn { A } { pi/2 }
```

```
\fp_show:n { \l_tmpb_fp }
\fp_show:N \l_tmpb_fp
\fp_set_variable:nn { A } { 0 }
\fp_show:n { \l_tmpb_fp }
\fp_show:N \l_tmpb_fp
```

defines `A` to be a variable, then defines `\l_tmpb_fp` to stand for `1*sin(A)+9` (note that `3**2` is evaluated, but the `1*` product is not simplified away). Until `\l_tmpb_fp` is changed, `\fp_show:N \l_tmpb_fp` will show `((1*sin(A))+9)` regardless of the value of `A`. The next step defines `A` to be equal to `pi/2`: then `\fp_show:n { \l_tmpb_fp }` will evaluate `\l_tmpb_fp` and show `10`. We then redefine `A` to be `0`: since `\l_tmpb_fp` still stands for `1*sin(A)+9`, the value shown is then `9`. Variables can be set with `\fp_set_variable:nn` to arbitrary floating point expressions including other variables.

`\fp_new_variable:n`

`\fp_new_variable:n {⟨identifier⟩}`

New: 2023-10-19

Declares the ⟨*identifier*⟩ as a variable, which allows it to be used in floating point expressions. For instance,

```
\fp_new_variable:n { A }
\fp_show:n { A**2 - A + 1 }
```

shows `(((A^2)-A)+1)`. If the declaration was missing, the parser would complain about an "`Unknown fp word 'A'`". The ⟨*identifier*⟩ must consist entirely of Latin letters among `[a-zA-Z]`.

`\fp_set_variable:nn`

`\fp_set_variable:nn {⟨identifier⟩} {⟨fp expr⟩}`

New: 2023-10-19

Defines the ⟨*identifier*⟩ to stand in any further expression for the result of evaluating the ⟨*floating point expression*⟩ as much as possible. The result may contain other variables, which are then replaced by their values if they have any. For instance,

```
\fp_new_variable:n { A }
\fp_new_variable:n { B }
\fp_new_variable:n { C }
\fp_set_variable:nn { A } { 3 }
\fp_set_variable:nn { C } { A ** 2 + B * 1 }
\fp_show:n { C + 4 }
\fp_set_variable:nn { A } { 4 }
\fp_show:n { C + 4 }
```

shows `((9+(B*1))+4)` twice: changing the value of `A` to `4` does not alter `C` because `A` was replaced by its value `3` when evaluating `A**2+B*1`.

`\fp_clear_variable:n` `\fp_clear_variable:n {⟨identifier⟩}`

New: 2023-10-19

Removes any value given by `\fp_set_variable:nn` to the variable with this ⟨*identifier*⟩. For instance,

```
\fp_new_variable:n { A }
\fp_set_variable:nn { A } { 3 }
\fp_show:n { A ^ 2 }
\fp_clear_variable:n { A }
\fp_show:n { A ^ 2 }
```

shows `9`, then `(A^2)`.

29.7 User-defined functions

It is possible to define new user functions which can be used inside the argument to `\fp_eval:n`, etc. These functions may take one or more named arguments, and should be implemented using expansion methods only.

`\fp_new_function:n` `\fp_new_function:n {⟨identifier⟩}`

New: 2023-10-19

Declares the ⟨*identifier*⟩ as a function, which allows it to be used in floating point expressions. For instance,

```
\fp_new_function:n { foo }
\fp_show:n { foo ( 1 + 2 , foo(3), A ) ** 2 } }
```

shows `(foo(3, foo(3), A))^(2)`. If the declaration was missing, the parser would complain about an "`Unknown fp word 'foo'`". The ⟨*identifier*⟩ must consist entirely of Latin letters `[a-zA-Z]`.

`\fp_set_function:nnn` `\fp_set_function:nnn {⟨identifier⟩} {⟨vars⟩} {⟨fp expr⟩}`

New: 2023-10-19

Defines the ⟨*identifier*⟩ to stand in any further expression for the result of evaluating the ⟨*floating point expression*⟩, with the ⟨*identifier*⟩ accepting the ⟨*vars*⟩ (a non-empty comma-separated list). The result may contain other functions, which are then replaced by their results if they have any. For instance,

```
\fp_new_function:n { npow }
\fp_set_function:nnn { npow } { a,b } { a**b }
\fp_show:n { npow(16,0.25) }
```

shows `2`. The names of the ⟨*vars*⟩ must consist entirely of Latin letters `[a-zA-Z]`, but are otherwise not restricted: in particular, they are independent of any variables declared by `\fp_new_variable:n`.

`\fp_clear_function:n` `\fp_clear_function:n {⟨identifier⟩}`

New: 2023-10-19

Removes any definition given by `\fp_set_function:nnn` to the function with this ⟨*identifier*⟩.

29.8 Some useful constants, and scratch variables

`\c_zero_fp`
`\c_minus_zero_fp`

New: 2012-05-08

Zero, with either sign.

`\c_one_fp`

New: 2012-05-08

One as an `fp`: useful for comparisons in some places.

`\c_inf_fp`
`\c_minus_inf_fp`

New: 2012-05-08

Infinity, with either sign. These can be input directly in a floating point expression as `inf` and `-inf`.

`\c_nan_fp`

New: 2012-05-08

Not a number. This can be input directly in a floating point expression as `nan`.

`\c_e_fp`

Updated: 2012-05-08

The value of the base of the natural logarithm, $\mathrm{e} = \exp(1)$.

`\c_pi_fp`

Updated: 2013-11-17

The value of π. This can be input directly in a floating point expression as `pi`.

`\c_one_degree_fp`

New: 2012-05-08
Updated: 2013-11-17

The value of 1° in radians. Multiply an angle given in degrees by this value to obtain a result in radians. Note that trigonometric functions expecting an argument in radians or in degrees are both available. Within floating point expressions, this can be accessed as `deg`.

29.9 Scratch variables

`\l_tmpa_fp`
`\l_tmpb_fp`

Scratch floating points for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_fp`
`\g_tmpb_fp`

Scratch floating points for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

29.10 Floating point exceptions

The functions defined in this section are experimental, and their functionality may be altered or removed altogether.

"Exceptions" may occur when performing some floating point operations, such as `0 / 0`, or `10 ** 1e9999`. The relevant IEEE standard defines 5 types of exceptions, of which we implement 4.

- *Overflow* occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in $\pm\infty$.
- *Underflow* occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in ± 0.
- *Invalid operation* occurs for operations with no defined outcome, for instance $0/0$ or $\sin(\infty)$, and results in a `nan`. It also occurs for conversion functions whose target type does not have the appropriate infinite or `nan` value (*e.g.*, `\fp_to_dim:n`).
- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating functions at poles, *e.g.*, $\ln(0)$ or $\cot(0)$. This results in $\pm\infty$.

(not yet) *Inexact* occurs whenever the result of a computation is not exact, in other words, almost always. At the moment, this exception is entirely ignored in LaTeX3.

To each exception we associate a "flag": `\l_fp_overflow_flag`, `\l_fp_underflow_flag`, `\l_fp_invalid_operation_flag` and `\l_fp_division_by_zero_flag`. The state of these flags can be tested and modified with commands from l3flag

By default, the "invalid operation" exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions raise the corresponding flag but do not trigger an error. The behaviour when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and raise the flag, or only raise the flag, or do nothing at all.

`\fp_trap:nn`

New: 2012-07-19
Updated: 2017-02-13

`\fp_trap:nn {⟨exception⟩} {⟨trap type⟩}`

All occurrences of the ⟨*exception*⟩ (`overflow`, `underflow`, `invalid_operation` or `division_by_zero`) within the current group are treated as ⟨*trap type*⟩, which can be

- `none`: the ⟨*exception*⟩ will be entirely ignored, and leave no trace;
- `flag`: the ⟨*exception*⟩ will turn the corresponding flag on when it occurs;
- `error`: additionally, the ⟨*exception*⟩ will halt the TeX run and display some information about the current operation in the terminal.

`\l_fp_overflow_flag`
`\l_fp_underflow_flag`
`\l_fp_invalid_operation_flag`
`\l_fp_division_by_zero_flag`

Flags denoting the occurrence of various floating-point exceptions.

29.11 Viewing floating points

`\fp_show:N`
`\fp_show:c`
`\fp_show:n`

New: 2012-05-08
Updated: 2021-04-29

`\fp_show:N` ⟨*fp var*⟩
`\fp_show:n {`⟨*fp expr*⟩`}`

Evaluates the ⟨*fp expr*⟩ and displays the result in the terminal.

`\fp_log:N`
`\fp_log:c`
`\fp_log:n`

New: 2014-08-22
Updated: 2021-04-29

`\fp_log:N` ⟨*fp var*⟩
`\fp_log:n {`⟨*fp expr*⟩`}`

Evaluates the ⟨*fp expr*⟩ and writes the result in the log file.

29.12 Floating point expressions

29.12.1 Input of floating point numbers

We support four types of floating point numbers:

- $\pm m \cdot 10^n$, a floating point number, with integer $1 \leq m \leq 10^{16}$, and $-10000 \leq n \leq 10000$;
- ± 0, zero, with a given sign;
- $\pm\infty$, infinity, with a given sign;
- `nan`, is "not a number", and can be either quiet or signalling (*not yet*: this distinction is currently unsupported);

Normal floating point numbers are stored in base 10, with up to 16 significant figures.

On input, a normal floating point number consists of:

- ⟨*sign*⟩: a possibly empty string of `+` and `-` characters;
- ⟨*significand*⟩: a non-empty string of digits together with zero or one dot;
- ⟨*exponent*⟩ optionally: the character `e` or `E`, followed by a possibly empty string of `+` and `-` tokens, and a non-empty string of digits.

The sign of the resulting number is `+` if ⟨*sign*⟩ contains an even number of `-`, and `-` otherwise, hence, an empty ⟨*sign*⟩ denotes a non-negative input. The stored significand is obtained from ⟨*significand*⟩ by omitting the decimal separator and leading zeros, and rounding to 16 significant digits, filling with trailing zeros if necessary. In particular, the value stored is exact if the input ⟨*significand*⟩ has at most 16 digits. The stored ⟨*exponent*⟩ is obtained by combining the input ⟨*exponent*⟩ (0 if absent) with a shift depending on the position of the significand and the number of leading zeros.

A special case arises if the resulting ⟨*exponent*⟩ is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by $\pm\infty$), or an underflow (resulting in ± 0).

The result is thus ± 0 if and only if $\langle$*significand*$\rangle$ contains no non-zero digit (*i.e.*, consists only in characters `0`, and an optional period), or if there is an underflow. Note that a single dot is currently a valid floating point number, equal to $+0$, but that is not guaranteed to remain true.

The $\langle$*significand*$\rangle$ must be non-empty, so `e1` and `e-1` are not valid floating point numbers. Note that the latter could be mistaken with the difference of "`e`" and 1. To avoid confusions, the base of natural logarithms cannot be input as `e` and should be input as `exp(1)` or `\c_e_fp` (which is faster).

Special numbers are input as follows:

- `inf` represents $+\infty$, and can be preceded by any $\langle$*sign*$\rangle$, yielding $\pm\infty$ as appropriate.
- `nan` represents a (quiet) non-number. It can be preceded by any sign, but that sign is ignored.
- Any unrecognizable string triggers an error, and produces a `nan`.
- Note that commands such as `\infty`, `\pi`, or `\sin` *do not* work in floating point expressions. They may silently be interpreted as completely unexpected numbers, because integer constants (allowed in expressions) are commonly stored as mathematical characters.

29.12.2 Precedence of operators

We list here all the operations supported in floating point expressions, in order of decreasing precedence: operations listed earlier bind more tightly than operations listed below them.

- Function calls (`sin`, `ln`, *etc*).
- Binary `**` and `^` (right associative).
- Unary `+`, `-`, `!`.
- Implicit multiplication by juxtaposition (`2pi`) when neither factor is in parentheses.
- Binary `*` and `/`, implicit multiplication by juxtaposition with parentheses (for instance `3(4+5)`).
- Binary `+` and `-`.
- Comparisons `>=`, `!=`, `<?`, *etc*.
- Logical `and`, denoted by `&&`.
- Logical `or`, denoted by `||`.
- Ternary operator `?:` (right associative).
- Comma (to build tuples).

The precedence of operations can be overridden using parentheses. In particular, the precedence of juxtaposition implies that

$$\begin{aligned}
\texttt{1/2pi} &= 1/(2\pi),\\
\texttt{1/2pi(pi + pi)} &= (2\pi)^{-1}(\pi+\pi) \simeq 1,\\
\texttt{sin2pi} &= \sin(2)\pi \neq 0,\\
\texttt{2\^{}2max(3,5)} &= 2^2 \max(3,5) = 20,\\
\texttt{1in/1cm} &= (1\mathrm{in})/(1\mathrm{cm}) = 2.54.
\end{aligned}$$

Functions are called on the value of their argument, contrarily to TeX macros.

29.12.3 Operations

We now present the various operations allowed in floating point expressions, from the lowest precedence to the highest. When used as a truth value, a floating point expression is `false` if it is ± 0, and `true` otherwise, including when it is `nan` or a tuple such as $(0, 0)$. Tuples are only supported to some extent by operations that work with truth values (`?:`, `||`, `&&`, `!`), by comparisons (`!<=>?`), and by `+`, `-`, `*`, `/`. Unless otherwise specified, providing a tuple as an argument of any other operation yields the "invalid operation" exception and a `nan` result.

`?:` `\fp_eval:n { ⟨operand1⟩ ? ⟨operand2⟩ : ⟨operand3⟩ }`

The ternary operator `?:` results in ⟨*operand*$_2$⟩ if ⟨*operand*$_1$⟩ is true (not ± 0), and ⟨*operand*$_3$⟩ if ⟨*operand*$_1$⟩ is false (± 0). All three ⟨*operands*⟩ are evaluated in all cases; they may be tuples. The operator is right associative, hence

```
\fp_eval:n
  {
    1 + 3 > 4 ? 1 :
    2 + 4 > 5 ? 2 :
    3 + 5 > 6 ? 3 : 4
  }
```

first tests whether $1 + 3 > 4$; since this isn't true, the branch following `:` is taken, and $2+4 > 5$ is compared; since this is true, the branch before `:` is taken, and everything else is (evaluated then) ignored. That allows testing for various cases in a concise manner, with the drawback that all computations are made in all cases.

`||` `\fp_eval:n { ⟨operand1⟩ || ⟨operand2⟩ }`

If ⟨*operand*$_1$⟩ is true (not ± 0), use that value, otherwise the value of ⟨*operand*$_2$⟩. Both ⟨*operands*⟩ are evaluated in all cases; they may be tuples. In ⟨*operand*$_1$⟩ `||` ⟨*operand*$_2$⟩ `||` ... `||` ⟨*operands*$_n$⟩, the first true (nonzero) ⟨*operand*⟩ is used and if all are zero the last one (± 0) is used.

`&&` `\fp_eval:n { ⟨operand1⟩ && ⟨operand2⟩ }`

If ⟨*operand*$_1$⟩ is false (equal to ± 0), use that value, otherwise the value of ⟨*operand*$_2$⟩. Both ⟨*operands*⟩ are evaluated in all cases; they may be tuples. In ⟨*operand*$_1$⟩ `&&` ⟨*operand*$_2$⟩ `&&` ... `&&` ⟨*operands*$_n$⟩, the first false (± 0) ⟨*operand*⟩ is used and if none is zero the last one is used.

<
=
>
?

Updated: 2013-12-14

```
\fp_eval:n
  {
    ⟨operand1⟩ ⟨relation1⟩
    ...
    ⟨operandN⟩ ⟨relationN⟩
    ⟨operandN+1⟩
  }
```

Each ⟨*relation*⟩ consists of a non-empty string of `<`, `=`, `>`, and `?`, optionally preceded by `!`, and may not start with `?`. This evaluates to $+1$ if all comparisons ⟨*operand*$_i$⟩ ⟨*relation*$_i$⟩ ⟨*operand*$_{i+1}$⟩ are true, and $+0$ otherwise. All ⟨*operands*⟩ are evaluated (once) in all cases. See `\fp_compare:nTF` for details.

+ `\fp_eval:n { ⟨operand1⟩ + ⟨operand2⟩ }`
- `\fp_eval:n { ⟨operand1⟩ - ⟨operand2⟩ }`

Computes the sum or the difference of its two ⟨*operands*⟩. The "invalid operation" exception occurs for $\infty - \infty$. "Underflow" and "overflow" occur when appropriate. These operations supports the itemwise addition or subtraction of two tuples, but if they have a different number of items the "invalid operation" exception occurs and the result is `nan`.

* `\fp_eval:n { ⟨operand1⟩ * ⟨operand2⟩ }`
/ `\fp_eval:n { ⟨operand1⟩ / ⟨operand2⟩ }`

Computes the product or the ratio of its two ⟨*operands*⟩. The "invalid operation" exception occurs for ∞/∞, $0/0$, or $0 * \infty$. "Division by zero" occurs when dividing a finite non-zero number by ± 0. "Underflow" and "overflow" occur when appropriate. When ⟨*operand*$_1$⟩ is a tuple and ⟨*operand*$_2$⟩ is a floating point number, each item of ⟨*operand*$_1$⟩ is multiplied or divided by ⟨*operand*$_2$⟩. Multiplication also supports the case where ⟨*operand*$_1$⟩ is a floating point number and ⟨*operand*$_2$⟩ a tuple. Other combinations yield an "invalid operation" exception and a `nan` result.

+ `\fp_eval:n { + ⟨operand⟩ }`
- `\fp_eval:n { - ⟨operand⟩ }`
! `\fp_eval:n { ! ⟨operand⟩ }`

The unary `+` does nothing, the unary `-` changes the sign of the ⟨*operand*⟩ (for a tuple, of all its components), and `!` ⟨*operand*⟩ evaluates to 1 if ⟨*operand*⟩ is false (is ± 0) and 0 otherwise (this is the `not` boolean function). Those operations never raise exceptions.

** `\fp_eval:n { ⟨operand1⟩ ** ⟨operand2⟩ }`
^ `\fp_eval:n { ⟨operand1⟩ ^ ⟨operand2⟩ }`

Raises ⟨*operand*$_1$⟩ to the power ⟨*operand*$_2$⟩. This operation is right associative, hence `2 ** 2 ** 3` equals $2^{2^3} = 256$. If ⟨*operand*$_1$⟩ is negative or -0 then: the result's sign is $+$ if the ⟨*operand*$_2$⟩ is infinite and $(-1)^p$ if the ⟨*operand*$_2$⟩ is $p/5^q$ with p, q integers; the result is $+0$ if `abs(`⟨*operand*$_1$⟩`)^`⟨*operand*$_2$⟩ evaluates to zero; in other cases the "invalid operation" exception occurs because the sign cannot be determined. "Division by zero" occurs when raising ± 0 to a finite strictly negative power. "Underflow" and "overflow" occur when appropriate. If either operand is a tuple, "invalid operation" occurs.

abs `\fp_eval:n { abs( ⟨fp expr⟩ ) }`

Computes the absolute value of the ⟨*fp expr*⟩. If the operand is a tuple, "invalid operation" occurs. This operation does not raise exceptions in other cases. See also `\fp_abs:n`.

`exp` `\fp_eval:n { exp(` ⟨*fp expr*⟩ `) }`

Computes the exponential of the ⟨*fp expr*⟩. "Underflow" and "overflow" occur when appropriate. If the operand is a tuple, "invalid operation" occurs.

`fact` `\fp_eval:n { fact(` ⟨*fp expr*⟩ `) }`

Computes the factorial of the ⟨*fp expr*⟩. If the ⟨*fp expr*⟩ is an integer between -0 and 3248 included, the result is finite and correctly rounded. Larger positive integers give $+\infty$ with "overflow", while `fact`$(+\infty) = +\infty$ and `fact(nan)` = `nan` with no exception. All other inputs give `nan` with the "invalid operation" exception.

`ln` `\fp_eval:n { ln(` ⟨*fp expr*⟩ `) }`

Computes the natural logarithm of the ⟨*fp expr*⟩. Negative numbers have no (real) logarithm, hence the "invalid operation" is raised in that case, including for $\ln(-0)$. "Division by zero" occurs when evaluating $\ln(+0) = -\infty$. "Underflow" and "overflow" occur when appropriate. If the operand is a tuple, "invalid operation" occurs.

`logb` ⋆ `\fp_eval:n { logb(` ⟨*fp expr*⟩ `) }`

New: 2018-11-03

Determines the exponent of the ⟨*fp expr*⟩, namely the floor of the base-10 logarithm of its absolute value. "Division by zero" occurs when evaluating $\text{logb}(\pm 0) = -\infty$. Other special values are $\text{logb}(\pm\infty) = +\infty$ and logb(`nan`) = `nan`. If the operand is a tuple or is `nan`, then "invalid operation" occurs and the result is `nan`.

`max` `\fp_eval:n { max(` ⟨*fp expr*$_1$⟩ `,` ⟨*fp expr*$_2$⟩ `, ... ) }`
`min` `\fp_eval:n { min(` ⟨*fp expr*$_1$⟩ `,` ⟨*fp expr*$_2$⟩ `, ... ) }`

Evaluates each ⟨*fp expr*⟩ and computes the largest (smallest) of those. If any of the ⟨*fp expr*⟩ is a `nan` or tuple, the result is `nan`. If any operand is a tuple, "invalid operation" occurs; these operations do not raise exceptions in other cases.

round
trunc
ceil
floor

New: 2013-12-14
Updated: 2015-08-08

```
\fp_eval:n { round ( ⟨fp expr⟩ ) }
\fp_eval:n { round ( ⟨fp expr1⟩ , ⟨fp expr2⟩ ) }
\fp_eval:n { round ( ⟨fp expr1⟩ , ⟨fp expr2⟩ , ⟨fp expr3⟩ ) }
```

Only `round` accepts a third argument. Evaluates ⟨*fp expr*$_1$⟩ $= x$ and ⟨*fp expr*$_2$⟩ $= n$ and ⟨*fp expr*$_3$⟩ $= t$ then rounds x to n places. If n is an integer, this rounds x to a multiple of 10^{-n}; if $n = +\infty$, this always yields x; if $n = -\infty$, this yields one of ± 0, $\pm\infty$, or `nan`; if $n =$ `nan`, this yields `nan`; if n is neither $\pm\infty$ nor an integer, then an "invalid operation" exception is raised. When ⟨*fp expr*$_2$⟩ is omitted, $n = 0$, *i.e.*, ⟨*fp expr*$_1$⟩ is rounded to an integer. The rounding direction depends on the function.

- `round` yields the multiple of 10^{-n} closest to x, with ties (x half-way between two such multiples) rounded as follows. If t is `nan` (or not given) the even multiple is chosen ("ties to even"), if $t = \pm 0$ the multiple closest to 0 is chosen ("ties to zero"), if t is positive/negative the multiple closest to $\infty/-\infty$ is chosen ("ties towards positive/negative infinity").
- `floor` yields the largest multiple of 10^{-n} smaller or equal to x ("round towards negative infinity");
- `ceil` yields the smallest multiple of 10^{-n} greater or equal to x ("round towards positive infinity");
- `trunc` yields a multiple of 10^{-n} with the same sign as x and with the largest absolute value less than that of x ("round towards zero").

"Overflow" occurs if x is finite and the result is infinite (this can only happen if ⟨*fp expr*$_2$⟩ < -9984). If any operand is a tuple, "invalid operation" occurs.

sign

```
\fp_eval:n { sign( ⟨fp expr⟩ ) }
```

Evaluates the ⟨*fp expr*⟩ and determines its sign: $+1$ for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ± 0 for ± 0, and `nan` for `nan`. If the operand is a tuple, "invalid operation" occurs. This operation does not raise exceptions in other cases.

sin
cos
tan
cot
csc
sec

Updated: 2013-11-17

```
\fp_eval:n { sin( ⟨fp expr⟩ ) }
\fp_eval:n { cos( ⟨fp expr⟩ ) }
\fp_eval:n { tan( ⟨fp expr⟩ ) }
\fp_eval:n { cot( ⟨fp expr⟩ ) }
\fp_eval:n { csc( ⟨fp expr⟩ ) }
\fp_eval:n { sec( ⟨fp expr⟩ ) }
```

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the ⟨*fp expr*⟩ given in radians. For arguments given in degrees, see `sind`, `cosd`, *etc.* Note that since π is irrational, sin(8pi) is not quite zero, while its analogue sind(8×180) is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the "invalid operation" exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a "division by zero" exception. "Underflow" and "overflow" occur when appropriate. If the operand is a tuple, "invalid operation" occurs.

sind
cosd
tand
cotd
cscd
secd

```
\fp_eval:n { sind( ⟨fp expr⟩ ) }
\fp_eval:n { cosd( ⟨fp expr⟩ ) }
\fp_eval:n { tand( ⟨fp expr⟩ ) }
\fp_eval:n { cotd( ⟨fp expr⟩ ) }
\fp_eval:n { cscd( ⟨fp expr⟩ ) }
\fp_eval:n { secd( ⟨fp expr⟩ ) }
```

New: 2013-11-02

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the ⟨*fp expr*⟩ given in degrees. For arguments given in radians, see `sin`, `cos`, *etc.* Note that since π is irrational, sin(8pi) is not quite zero, while its analogue sind(8×180) is exactly zero. The trigonometric functions are undefined for an argument of $\pm\infty$, leading to the "invalid operation" exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a "division by zero" exception. "Underflow" and "overflow" occur when appropriate. If the operand is a tuple, "invalid operation" occurs.

asin
acos
acsc
asec

```
\fp_eval:n { asin( ⟨fp expr⟩ ) }
\fp_eval:n { acos( ⟨fp expr⟩ ) }
\fp_eval:n { acsc( ⟨fp expr⟩ ) }
\fp_eval:n { asec( ⟨fp expr⟩ ) }
```

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the ⟨*fp expr*⟩ and returns the result in radians, in the range $[-\pi/2, \pi/2]$ for `asin` and `acsc` and $[0, \pi]$ for `acos` and `asec`. For a result in degrees, use `asind`, *etc.* If the argument of `asin` or `acos` lies outside the range $[-1, 1]$, or the argument of `acsc` or `asec` inside the range $(-1, 1)$, an "invalid operation" exception is raised. "Underflow" and "overflow" occur when appropriate. If the operand is a tuple, "invalid operation" occurs.

asind
acosd
acscd
asecd

```
\fp_eval:n { asind( ⟨fp expr⟩ ) }
\fp_eval:n { acosd( ⟨fp expr⟩ ) }
\fp_eval:n { acscd( ⟨fp expr⟩ ) }
\fp_eval:n { asecd( ⟨fp expr⟩ ) }
```

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the ⟨*fp expr*⟩ and returns the result in degrees, in the range $[-90, 90]$ for `asind` and `acscd` and $[0, 180]$ for `acosd` and `asecd`. For a result in radians, use `asin`, *etc.* If the argument of `asind` or `acosd` lies outside the range $[-1, 1]$, or the argument of `acscd` or `asecd` inside the range $(-1, 1)$, an "invalid operation" exception is raised. "Underflow" and "overflow" occur when appropriate. If the operand is a tuple, "invalid operation" occurs.

atan
acot

New: 2013-11-02

```
\fp_eval:n { atan( ⟨fp expr⟩ ) }
\fp_eval:n { atan( ⟨fp expr1⟩ , ⟨fp expr2⟩ ) }
\fp_eval:n { acot( ⟨fp expr⟩ ) }
\fp_eval:n { acot( ⟨fp expr1⟩ , ⟨fp expr2⟩ ) }
```

Those functions yield an angle in radians: `atand` and `acotd` are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the ⟨*fp expr*⟩: arctangent takes values in the range $[-\pi/2, \pi/2]$, and arccotangent in the range $[0, \pi]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates (⟨*fp expr*$_2$⟩, ⟨*fp expr*$_1$⟩): this is the arctangent of ⟨*fp expr*$_1$⟩/⟨*fp expr*$_2$⟩, possibly shifted by π depending on the signs of ⟨*fp expr*$_1$⟩ and ⟨*fp expr*$_2$⟩. The two-argument arccotangent computes the angle in polar coordinates of the point (⟨*fp expr*$_1$⟩, ⟨*fp expr*$_2$⟩), equal to the arccotangent of ⟨*fp expr*$_1$⟩/⟨*fp expr*$_2$⟩, possibly shifted by π. Both two-argument functions take values in the wider range $[-\pi, \pi]$. The ratio ⟨*fp expr*$_1$⟩/⟨*fp expr*$_2$⟩ need not be defined for the two-argument arctangent: when both expressions yield ± 0, or when both yield $\pm\infty$, the resulting angle is one of $\{\pm\pi/4, \pm 3\pi/4\}$ depending on signs. The "underflow" exception can occur. If any operand is a tuple, "invalid operation" occurs.

atand
acotd

New: 2013-11-02

```
\fp_eval:n { atand( ⟨fp expr⟩ ) }
\fp_eval:n { atand( ⟨fp expr1⟩ , ⟨fp expr2⟩ ) }
\fp_eval:n { acotd( ⟨fp expr⟩ ) }
\fp_eval:n { acotd( ⟨fp expr1⟩ , ⟨fp expr2⟩ ) }
```

Those functions yield an angle in degrees: `atan` and `acot` are their analogs in radians. The one-argument versions compute the arctangent or arccotangent of the ⟨*fp expr*⟩: arctangent takes values in the range $[-90, 90]$, and arccotangent in the range $[0, 180]$. The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates (⟨*fp expr*$_2$⟩, ⟨*fp expr*$_1$⟩): this is the arctangent of ⟨*fp expr*$_1$⟩/⟨*fp expr*$_2$⟩, possibly shifted by 180 depending on the signs of ⟨*fp expr*$_1$⟩ and ⟨*fp expr*$_2$⟩. The two-argument arccotangent computes the angle in polar coordinates of the point (⟨*fp expr*$_1$⟩, ⟨*fp expr*$_2$⟩), equal to the arccotangent of ⟨*fp expr*$_1$⟩/⟨*fp expr*$_2$⟩, possibly shifted by 180. Both two-argument functions take values in the wider range $[-180, 180]$. The ratio ⟨*fp expr*$_1$⟩/⟨*fp expr*$_2$⟩ need not be defined for the two-argument arctangent: when both expressions yield ± 0, or when both yield $\pm\infty$, the resulting angle is one of $\{\pm 45, \pm 135\}$ depending on signs. The "underflow" exception can occur. If any operand is a tuple, "invalid operation" occurs.

sqrt

New: 2013-12-14

```
\fp_eval:n { sqrt( ⟨fp expr⟩ ) }
```

Computes the square root of the ⟨*fp expr*⟩. The "invalid operation" is raised when the ⟨*fp expr*⟩ is negative or is a tuple; no other exception can occur. Special values yield $\sqrt{-0} = -0$, $\sqrt{+0} = +0$, $\sqrt{+\infty} = +\infty$ and $\sqrt{\texttt{nan}} = \texttt{nan}$.

rand

New: 2016-12-05

`\fp_eval:n { rand() }`

Produces a pseudo-random floating-point number (multiple of 10^{-16}) between 0 included and 1 excluded. This is not available in older versions of XeTeX. The random seed can be queried using `\sys_rand_seed:` and set using `\sys_gset_rand_seed:n`.

TeXhackers note: This is based on pseudo-random numbers provided by the engine's primitive `\pdfuniformdeviate` in pdfTeX, pTeX, upTeX and `\uniformdeviate` in LuaTeX and XeTeX. The underlying code is based on Metapost, which follows an additive scheme recommended in Section 3.6 of "The Art of Computer Programming, Volume 2".

While we are more careful than `\uniformdeviate` to preserve uniformity of the underlying stream of 28-bit pseudo-random integers, these pseudo-random numbers should of course not be relied upon for serious numerical computations nor cryptography.

randint

New: 2016-12-05

`\fp_eval:n { randint(` ⟨*fp expr*⟩ `) }`
`\fp_eval:n { randint(` ⟨*fp expr*$_1$⟩ `,` ⟨*fp expr*$_2$⟩ `) }`

Produces a pseudo-random integer between 1 and ⟨*fp expr*⟩ or between ⟨*fp expr*$_1$⟩ and ⟨*fp expr*$_2$⟩ inclusive. The bounds must be integers in the range $(-10^{16}, 10^{16})$ and the first must be smaller or equal to the second. See `rand` for important comments on how these pseudo-random numbers are generated.

inf
nan

The special values $+\infty$, $-\infty$, and `nan` are represented as `inf`, `-inf` and `nan` (see `\c_inf_fp`, `\c_minus_inf_fp` and `\c_nan_fp`).

pi

The value of π (see `\c_pi_fp`).

deg

The value of 1° in radians (see `\c_one_degree_fp`).

em ex in pt pc cm mm dd cc nd nc bp sp

Those units of measurement are equal to their values in pt, namely

$$
\begin{aligned}
1\,\texttt{in} &= 72.27\,\texttt{pt}\\
1\,\texttt{pt} &= 1\,\texttt{pt}\\
1\,\texttt{pc} &= 12\,\texttt{pt}\\
1\,\texttt{cm} &= \frac{1}{2.54}\,\texttt{in} = 28.45275590551181\,\texttt{pt}\\
1\,\texttt{mm} &= \frac{1}{25.4}\,\texttt{in} = 2.845275590551181\,\texttt{pt}\\
1\,\texttt{dd} &= 0.376065\,\texttt{mm} = 1.07000856496063\,\texttt{pt}\\
1\,\texttt{cc} &= 12\,\texttt{dd} = 12.84010277952756\,\texttt{pt}\\
1\,\texttt{nd} &= 0.375\,\texttt{mm} = 1.066978346456693\,\texttt{pt}\\
1\,\texttt{nc} &= 12\,\texttt{nd} = 12.80374015748031\,\texttt{pt}\\
1\,\texttt{bp} &= \frac{1}{72}\,\texttt{in} = 1.00375\,\texttt{pt}\\
1\,\texttt{sp} &= 2^{-16}\,\texttt{pt} = 1.52587890625\times 10^{-5}\,\texttt{pt}.
\end{aligned}
$$

The values of the (font-dependent) units em and ex are gathered from TeX when the surrounding floating point expression is evaluated.

true false

Other names for 1 and +0.

\fp_abs:n ⋆

New: 2012-05-14
Updated: 2012-07-08

\fp_abs:n {⟨*fp expr*⟩}

Evaluates the ⟨*fp expr*⟩ as described for \fp_eval:n and leaves the absolute value of the result in the input stream. If the argument is $\pm\infty$, nan or a tuple, "invalid operation" occurs. Within floating point expressions, abs() can be used; it accepts $\pm\infty$ and nan as arguments.

\fp_max:nn ⋆
\fp_min:nn ⋆

New: 2012-09-26

\fp_max:nn {⟨*fp expr*$_1$⟩} {⟨*fp expr*$_2$⟩}

Evaluates the ⟨*fp exprs*⟩ as described for \fp_eval:n and leaves the resulting larger (max) or smaller (min) value in the input stream. If the argument is a tuple, "invalid operation" occurs, but no other case raises exceptions. Within floating point expressions, max() and min() can be used.

29.13 Disclaimer and roadmap

This module may break if the escape character is among 0123456789_+, or if it receives a TeX primitive conditional affected by \exp_not:N.

The following need to be done. I'll try to time-order the items.

- Function to count items in a tuple (and to determine if something is a tuple).
- Decide what exponent range to consider.

- Support signalling `nan`.
- Modulo and remainder, and rounding function `quantize` (and its friends analogous to `trunc`, `ceil`, `floor`).
- `\fp_format:nn {`⟨*fp expr*⟩`} {`⟨*format*⟩`}`, but what should ⟨`format`⟩ be? More general pretty printing?
- Add `and`, `or`, `xor`? Perhaps under the names `all`, `any`, and `xor`?
- Add $\log(x, b)$ for logarithm of x in base b.
- `hypot` (Euclidean length). Cartesian-to-polar transform.
- Hyperbolic functions `cosh`, `sinh`, `tanh`.
- Inverse hyperbolics.
- Base conversion, input such as `0xAB.CDEF`.
- Factorial (not with `!`), gamma function.
- Improve coefficients of the `sin` and `tan` series.
- Treat upper and lower case letters identically in identifiers, and ignore underscores.
- Add an `array(1,2,3)` and `i=complex(0,1)`.
- Provide an experimental `map` function? Perhaps easier to implement if it is a single character, `@sin(1,2)`?
- Provide an `isnan` function analogue of `\fp_if_nan:nTF`?
- Support keyword arguments?

Pgfmath also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs, and tests to add.

- Check that functions are monotonic when they should.
- Add exceptions to `?:`, `!<=>?`, `&&`, `||`, and `!`.
- Logarithms of numbers very close to 1 are inaccurate.
- When rounding towards $-\infty$, `\dim_to_fp:n {0pt}` should return -0, not $+0$.
- The result of $(\pm 0) + (\pm 0)$, of $x + (-x)$, and of $(-x) + x$ should depend on the rounding mode.
- `0e9999999999` gives a TeX "number too large" error.
- Subnormals are not implemented.

Possible optimizations/improvements.

- Document that l3trial/l3fp-types introduces tools for adding new types.
- In subsection 29.12.1, write a grammar.

- It would be nice if the `parse` auxiliaries for each operation were set up in the corresponding module, rather than centralizing in l3fp-parse.
- Some functions should get an `_o` ending to indicate that they expand after their result.
- More care should be given to distinguish expandable/restricted expandable (auxiliary and internal) functions.
- The code for the `ternary` set of functions is ugly.
- There are many `~` missing in the doc to avoid bad line-breaks.
- The algorithm for computing the logarithm of the significand could be made to use a 5 terms Taylor series instead of 10 terms by taking $c = 2000/(\lfloor 200x \rfloor + 1) \in [10, 95]$ instead of $c \in [1, 10]$. Also, it would then be possible to simplify the computation of t. However, we would then have to hard-code the logarithms of 44 small integers instead of 9.
- Improve notations in the explanations of the division algorithm (l3fp-basics).
- Understand and document `\__fp_basics_pack_weird_low:NNNNw` and `\__fp_basics_pack_weird_high:NNNNNNNNw` better. Move the other `basics_pack` auxiliaries to l3fp-aux under a better name.
- Find out if underflow can really occur for trigonometric functions, and redoc as appropriate.
- Add bibliography. Some of Kahan's articles, some previous TeX fp packages, the international standards,...
- Also take into account the "inexact" exception?
- Support multi-character prefix operators (*e.g.*, `@/` or whatever)?

Chapter 30

The l3fparray module
Fast global floating point arrays

For applications requiring heavy use of floating points, this module provides arrays which can be accessed in constant time (contrast l3seq, where access time is linear). The interface is very close to that of l3intarray. The size of the array is fixed and must be given at point of initialisation

30.1 Creating and initialising floating point array variables

`\fparray_new:Nn`
`\fparray_new:cn`

New: 2018-05-05

`\fparray_new:Nn ⟨fparray var⟩ {⟨size⟩}`

Evaluates the integer expression ⟨*size*⟩ and allocates an ⟨*floating point array variable*⟩ with that number of (zero) entries. The variable name should start with `\g_` because assignments are always global.

`\fparray_gzero:N`
`\fparray_gzero:c`

New: 2018-05-05

`\fparray_gzero:N ⟨fparray var⟩`

Sets all entries of the ⟨*floating point array variable*⟩ to +0. Assignments are always global.

30.2 Adding data to floating point arrays

`\fparray_gset:Nnn`
`\fparray_gset:cnn`

New: 2018-05-05

`\fparray_gset:Nnn ⟨fparray var⟩ {⟨position⟩} {⟨value⟩}`

Stores the result of evaluating the floating point expression ⟨*value*⟩ into the ⟨*floating point array variable*⟩ at the (integer expression) ⟨*position*⟩. If the ⟨*position*⟩ is not between 1 and the `\fparray_count:N`, an error occurs. Assignments are always global.

30.3 Counting entries in floating point arrays

\fparray_count:N ⋆
\fparray_count:c ⋆

New: 2018-05-05

\fparray_count:N ⟨*fparray var*⟩

Expands to the number of entries in the ⟨*floating point array variable*⟩. This is performed in constant time.

30.4 Using a single entry

\fparray_item:Nn ⋆
\fparray_item:cn ⋆
\fparray_item_to_tl:Nn ⋆
\fparray_item_to_tl:cn ⋆

New: 2018-05-05

\fparray_item:Nn ⟨*fparray var*⟩ {⟨*position*⟩}

Applies \fp_use:N or \fp_to_tl:N (respectively) to the floating point entry stored at the (integer expression) ⟨*position*⟩ in the ⟨*floating point array variable*⟩. If the ⟨*position*⟩ is not between 1 and the \fparray_count:N ⟨*fparray var*⟩, an error occurs.

30.5 Floating point array conditional

\fparray_if_exist_p:N ⋆
\fparray_if_exist_p:c ⋆
\fparray_if_exist:N*TF* ⋆
\fparray_if_exist:c*TF* ⋆

New: 2024-03-31

\fparray_if_exist_p:N ⟨*fparray var*⟩
\fparray_if_exist:NTF ⟨*fparray var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*fparray var*⟩ is currently defined. This does not check that the ⟨*fparray var*⟩ really is a floating point array variable.

Chapter 31

The l3bitset module Bitsets

This module defines and implements the data type bitset, a vector of bits. The size of the vector may grow dynamically. Individual bits can be set and unset by names pointing to an index position. The names 1, 2, 3, ... are predeclared and point to the index positions 1, 2, 3,.... More names can be added and existing names can be changed. The index is like all other indices in expl3 modules *1-based*. A bitset can be output as binary number or—as needed e.g. in a PDF dictionary—as decimal (arabic) number. Currently only a small subset of the functions provided by the bitset package are implemented here, mainly the functions needed to use bitsets in PDF dictionaries.

The bitset is stored as a string (but one shouldn't rely on the internal representation) and so the vector size is theoretically unlimited, only restricted by TEX-memory. But the functions to set and clear bits use integer functions for the index so bitsets can't be longer than $2^{31} - 1$. The export function \bitset_to_arabic:N can use functions from the int module only if the largest index used for this bitset is smaller than 32, for longer bitsets fp is used and this is slower.

31.1 Creating bitsets

\bitset_new:N
\bitset_new:c
\bitset_new:Nn
\bitset_new:cn

New: 2023-11-15

```
\bitset_new:N  ⟨bitset var⟩
\bitset_new:Nn ⟨bitset var⟩
  {
    ⟨name1⟩ = ⟨index1⟩ ,
    ⟨name2⟩ = ⟨index2⟩ , ...
  }
```

Creates a new ⟨`bitset var`⟩ or raises an error if the name is already taken. The declaration is global. The ⟨`bitset var`⟩ is initially 0.

Bitsets are implemented as string variables consisting of `1`'s and `0`'s. The rightmost number is the index position 1, so the string variable can be viewed directly as the binary number. But one shouldn't rely on the internal representation, but use the dedicated `\bitset_to_bin:N` instead to get the binary number.

The name–index pairs given in the second argument of `\bitset_new:Nn` declares names for some indices, which can be used to set and unset bits. The names `1`, `2`, `3`, ... are predeclared and point to the index positions 1, 2, 3, ...

⟨`index...`⟩ should be a positive number or an ⟨`integer expression`⟩ which evaluates to a positive number. The expression is evaluated when the index is used, not at declaration time. The names ⟨`name...`⟩ should be unique. Using a number as name, e.g. `10=1`, is allowed, it then overwrites the predeclared name `10`, but the index position 10 can then only be reached if some other name for it exists, e.g. `ten=10`. It is not necessary to give every index a name, and an index can have more than one name. The named index can be extended or changed with the next function.

\bitset_addto_named_index:Nn

New: 2023-11-15

```
\bitset_addto_named_index:Nn ⟨bitset var⟩
  {
    ⟨name1⟩ = ⟨index1⟩ ,
    ⟨name2⟩ = ⟨index2⟩ , ...
  }
```

This extends or changes the name–index pairs for ⟨`bitset var`⟩ globally as described for `\bitset_new:Nn`.

For example after these settings

```
\bitset_new:Nn \l_pdfannot_F_bitset
  {
    Invisible      = 1,
    Hidden         = 2,
    Print          = 3,
    NoZoom         = 4,
    NoRotate       = 5,
    NoView         = 6,
    ReadOnly       = 7,
    Locked         = 8,
    ToggleNoView   = 9,
    LockedContents = 10
  }
\bitset_addto_named_index:Nn \l_pdfannot_F_bitset
  {
```

```
    print = 3
  }
```

it is possible to set bit 3 by using any of these alternatives:

```
\bitset_set_true:Nn \l_pdfannot_F_bitset {Print}
\bitset_set_true:Nn \l_pdfannot_F_bitset {print}
\bitset_set_true:Nn \l_pdfannot_F_bitset {3}
```

`\bitset_if_exist_p:N` ⋆
`\bitset_if_exist_p:c` ⋆
`\bitset_if_exist:NTF` ⋆
`\bitset_if_exist:cTF` ⋆

New: 2023-11-15

`\bitset_if_exist_p:N` ⟨*bitset var*⟩
`\bitset_if_exist:NTF` ⟨*bitset var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨`bitset var`⟩ exist.

31.2 Setting and unsetting bits

`\bitset_set_true:Nn`
`\bitset_set_true:cn`
`\bitset_gset_true:Nn`
`\bitset_gset_true:cn`

New: 2023-11-15

`\bitset_set_true:Nn` ⟨*bitset var*⟩ {⟨*name*⟩}

This sets the bit of the index position represented by {⟨*name*⟩} to 1. ⟨`name`⟩ should be either one of the predeclared names `1`, `2`, `3`, …, or one of the names added manually. Index position are 1-based. If needed the length of the bit vector is enlarged.

`\bitset_set_false:Nn`
`\bitset_set_false:cn`
`\bitset_gset_false:Nn`
`\bitset_gset_false:cn`

New: 2023-11-15

`\bitset_set_false:Nn` ⟨*bitset var*⟩ {⟨*name*⟩}

This unsets the bit of the index position represented by {⟨*name*⟩} (sets it to 0). ⟨`name`⟩ should be either one of the predeclared names `1`, `2`, `3`, …, or one of the names added manually. The index is 1-based. If the index position is larger than the current length of the bit vector nothing happens. If the leading (left most) bit is unset, zeros are not trimmed but stay in the bit vector and are still shown by `\bitset_show:N`.

`\bitset_clear:N`
`\bitset_clear:c`
`\bitset_gclear:N`
`\bitset_gclear:c`

New: 2023-11-15

`\bitset_clear:N` ⟨*bitset var*⟩

This resets the bitset to the initial state. The declared names are not changed.

31.3 Using bitsets

`\bitset_item:Nn` ⋆
`\bitset_item:cn` ⋆

New: 2023-11-15

`\bitset_item:Nn` ⟨*bitset var*⟩ {⟨*name*⟩}

`\bitset_item:Nn` outputs `1` if the bit with the index number represented by ⟨`name`⟩ is set and `0` otherwise. ⟨`name`⟩ is either one of the predeclared names `1`, `2`, `3`, …, or one of the names added manually.

\bitset_to_bin:N ⋆
\bitset_to_bin:c ⋆

New: 2023-11-15

\bitset_to_bin:N ⟨*bitset var*⟩

This leaves the current value of the bitset expressed as a binary (string) number in the input stream. If no bit has been set yet, the output is zero.

\bitset_to_arabic:N ⋆
\bitset_to_arabic:c ⋆

New: 2023-11-15

\bitset_to_arabic:N ⟨*bitset var*⟩

This leaves the current value of the bitset expressed as a decimal number in the input stream. If no bit has been set yet, the output is zero. The function uses `\int_from_bin:n` if the largest index that have been set or unset is smaller than 32, and a slower implementation based on `\fp_eval:n` otherwise.

\bitset_show:N
\bitset_show:c

New: 2023-11-15

\bitset_show:N ⟨*bitset var*⟩

Displays the binary and decimal values of the ⟨*bitset var*⟩ on the terminal.

\bitset_log:N
\bitset_log:c

New: 2023-11-15

\bitset_log:N ⟨*bitset var*⟩

Writes the binary and decimal values of the ⟨*bitset var*⟩ in the log file.

\bitset_show_named_index:N
\bitset_show_named_index:c

New: 2023-11-15

\bitset_show_named_index:N ⟨*bitset var*⟩

Displays declared name–index pairs of the ⟨*bitset var*⟩ on the terminal.

\bitset_log_named_index:N
\bitset_log_named_index:c

New: 2023-12-11

\bitset_log_named_index:N ⟨*bitset var*⟩

Writes declared name–index pairs of the ⟨*bitset var*⟩ in the log file.

Chapter 32

The l3cctab module Category code tables

A category code table enables rapid switching of all category codes in one operation. For LuaTeX, this is possible over the entire Unicode range. For other engines, only the 8-bit range (0–255) is covered by such tables. The implementation of category code tables in expl3 also saves and restores the TeX `\endlinechar` primitive value, meaning they could be used for example to implement `\ExplSyntaxOn`.

32.1 Creating and initialising category code tables

`\cctab_new:N`
`\cctab_new:c`

Updated: 2020-07-02

`\cctab_new:N` ⟨*category code table*⟩

Creates a new ⟨*category code table*⟩ variable or raises an error if the name is already taken. The declaration is global. The ⟨*category code table*⟩ is initialised with the codes as used by iniTeX.

`\cctab_const:Nn`
`\cctab_const:cn`

Updated: 2020-07-07

`\cctab_const:Nn` ⟨*category code table*⟩ {⟨*category code set up*⟩}

Creates a new ⟨*category code table*⟩, applies (in a group) the ⟨*category code set up*⟩ on top of iniTeX settings, then saves them globally as a constant table. The ⟨*category code set up*⟩ can include a call to `\cctab_select:N`.

`\cctab_gset:Nn`
`\cctab_gset:cn`

Updated: 2020-07-07

`\cctab_gset:Nn` ⟨*category code table*⟩ {⟨*category code set up*⟩}

Starting from the iniTeX category codes, applies (in a group) the ⟨*category code set up*⟩, then saves them globally in the ⟨*category code table*⟩. The ⟨*category code set up*⟩ can include a call to `\cctab_select:N`.

`\cctab_gsave_current:N`
`\cctab_gsave_current:c`

New: 2023-05-26

`\cctab_gsave_current:N` ⟨*category code table*⟩

Saves the current prevailing category codes in the ⟨*category code table*⟩.

32.2 Using category code tables

\cctab_begin:N
\cctab_begin:c

Updated: 2020-07-02

\cctab_begin:N ⟨*category code table*⟩

Switches locally the category codes in force to those stored in the ⟨*category code table*⟩. The prevailing codes before the function is called are added to a stack, for use with \cctab_end:. This function does not start a TeX group.

\cctab_end:

Updated: 2020-07-02

\cctab_end:

Ends the scope of a ⟨*category code table*⟩ started using \cctab_begin:N, returning the codes to those in force before the matching \cctab_begin:N was used. This must be used within the same TeX group (and at the same TeX group level) as the matching \cctab_begin:N.

\cctab_select:N
\cctab_select:c

New: 2020-05-19
Updated: 2020-07-02

\cctab_select:N ⟨*category code table*⟩

Selects the ⟨*category code table*⟩ for the scope of the current group. This is in particular useful in the ⟨*setup*⟩ arguments of \tl_set_rescan:Nnn, \tl_rescan:nn, \cctab_const:Nn, and \cctab_gset:Nn.

\cctab_item:Nn ⋆
\cctab_item:cn ⋆

New: 2021-05-10

\cctab_item:Nn ⟨*category code table*⟩ {⟨*int expr*⟩}

Determines the ⟨*character*⟩ with character code given by the ⟨*int expr*⟩ and expands to its category code specified by the ⟨*category code table*⟩.

32.3 Category code table conditionals

\cctab_if_exist_p:N ⋆
\cctab_if_exist_p:c ⋆
\cctab_if_exist:N*TF* ⋆
\cctab_if_exist:c*TF* ⋆

\cctab_if_exist_p:N ⟨*category code table*⟩
\cctab_if_exist:NTF ⟨*category code table*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*category code table*⟩ is currently defined. This does not check that the ⟨*category code table*⟩ really is a category code table.

32.4 Constant and scratch category code tables

\c_code_cctab

Updated: 2020-07-10

Category code table for the expl3 code environment; this does *not* include @, which is retained as an "other" character. Sets the \endlinechar value to 32 (a space).

\c_document_cctab

Updated: 2020-07-08

Category code table for a standard LaTeX document, as set by the LaTeX kernel. In particular, the upper-half of the 8-bit range will be set to "active" with pdfTeX *only*. No babel shorthands will be activated. Sets the \endlinechar value to 13 (normal line ending).

`\c_initex_cctab` Category code table as set up by iniTeX.

Updated: 2020-07-02

`\c_other_cctab` Category code table where all characters have category code 12 (other). Sets the `\endlinechar` value to -1.

Updated: 2020-07-02

`\c_str_cctab` Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space). Sets the `\endlinechar` value to -1.

Updated: 2020-07-02

`\g_tmpa_cctab`
`\g_tmpb_cctab` Scratch category code tables.

New: 2023-05-26

Part V

Text manipulation

Chapter 33

The l3unicode module Unicode support functions

This module provides Unicode-specific functions along with loading data from a range of Unicode Consortium files. Most of the code here is internal, but there are a small set of public functions. These work with Unicode ⟨*codepoints*⟩ and are designed to give usable results with both Unicode-aware and 8-bit engines.

`\codepoint_generate:nn` ⋆ `\codepoint_generate:nn {⟨codepoint⟩} {⟨catcode⟩}`

New: 2022-10-09
Updated: 2022-11-09

Generates one or more character tokens representing the ⟨*codepoint*⟩. With Unicode engines, exactly one character token will be generated, and this will have the ⟨*catcode*⟩ specified as the second argument:

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 10 (space)
- 11 (letter)
- 12 (other)
- 13 (active)

For 8-bit engines, between one and four character tokens will be produced: these will be the bytes of the UTF-8 representation of the ⟨*codepoint*⟩. For all codepoints outside of the classical ASCII range, the generated character tokens will be active (category code 13); for codepoints in the ASCII range, the given ⟨*catcode*⟩ will be used. To allow the result of this function to be used inside an expansion context, the result is protected by `\exp_not:n`.

TEXhackers note: Users of (u)pTEX note that these engines are treated as 8-bit in this context. In particular, for upTEX, irrespective of the `\kcatcode` of the ⟨*codepoint*⟩, any value outside the ASCII range will result in a series of active bytes being generated.

`\codepoint_str_generate:n` ⋆ `\codepoint_str_generate:n {⟨codepoint⟩}`

New: 2022-10-09

Generates one or more character tokens representing the ⟨*codepoint*⟩. With Unicode engines, exactly one character token will be generated. For 8-bit engines, between one and four character tokens will be produced: these will be the bytes of the UTF-8 representation of the ⟨*codepoint*⟩. All of the generated character tokens will be of category code 12, except any spaces (codepoint 32), which will be category code 10.

`\codepoint_to_category:n` ⋆ `\codepoint_to_category:n {⟨codepoint⟩}`

New: 2023-06-19

Expands to the Unicode general category identifier of the ⟨*codepoint*⟩. The general category identifier is a string made up of two letter characters, the first uppercase and the second lowercase. The uppercase letters divide codepoints into broader groups, which are then refined by the lowercase letter. For example, codepoints representing letters all have identifiers starting `L`, for example `Lu` (uppercase letter), `Lt` (titlecase letter), *etc.* Full details are available in the documentation provided by the Unicode Consortium: see https://www.unicode.org/reports/tr44/#General_Category_Values

`\codepoint_to_nfd:n` ⋆ `\codepoint_to_nfd:n {⟨codepoint⟩}`

New: 2022-10-09

Converts the ⟨*codepoint*⟩ to the Unicode Normalization Form Canonical Decomposition. The generated character(s) will have the current category code as they would if typed in directly for Unicode engines; for 8-bit engines, active characters are used for all codepoints outside of the ASCII range.

Chapter 34

The l3text module Text processing

This module deals with manipulation of (formatted) text; such material is comprised of a restricted set of token list content. The functions provided here concern conversion of textual content for example in case changing, generation of bookmarks and extraction to tags. All of the major functions operate by expansion. Begin-group and end-group tokens in the ⟨*text*⟩ are normalized and become `{` and `}`, respectively.

34.1 Expanding text

`\text_expand:n` ⋆

New: 2020-01-02
Updated: 2023-06-09

`\text_expand:n {⟨text⟩}`

Takes user input ⟨*text*⟩ and expands the content. Protected commands (typically formatting) are left in place, and no processing of math mode material (as delimited by pairs given in `\l_text_math_delims_tl` or as the argument to commands listed in `\l_text_math_arg_tl`) takes place. Commands which are neither engine- nor LaTeX-protected are expanded exhaustively. Any commands listed in `\l_text_expand_exclude_tl` are excluded from expansion, as are those in `\l_text_case_exclude_arg_tl` and `\l_text_math_arg_tl`.

`\text_declare_expand_equivalent:Nn`
`\text_declare_expand_equivalent:cn`

New: 2020-01-22

`\text_declare_expand_equivalent:Nn ⟨cmd⟩ {⟨replacement⟩}`

Declares that the ⟨*replacement*⟩ tokens should be used whenever the ⟨*cmd*⟩ (a single token) is encountered. The ⟨*replacement*⟩ tokens should be expandable. A token can be "replaced" by itself if the defined replacement wraps it in `\exp_not:n`, for example

```
\text_declare_expand_equivalent:Nn \' { \exp_not:n { \' } }
```

34.2 Case changing

`\text_lowercase:n` ⋆
`\text_uppercase:n` ⋆
`\text_titlecase_all:n` ⋆
`\text_titlecase_first:n` ⋆
`\text_lowercase:nn` ⋆
`\text_uppercase:nn` ⋆
`\text_titlecase_all:nn` ⋆
`\text_titlecase_first:nn` ⋆

New: 2019-11-20
Updated: 2023-07-08

`\text_uppercase:n {⟨tokens⟩}`
`\text_uppercase:nn {⟨BCP-47⟩} {⟨tokens⟩}`

Takes user input ⟨*text*⟩ first applies `\text_expand:n`, then transforms the case of character tokens as specified by the function name. The category code of letters are not changed by this process when Unicode engines are used; in 8-bit engines, case changed charters in the ASCII range will have the current prevailing category code, while those outside of it will be represented by active characters.

Upper- and lowercase have the obvious meanings. Titlecasing may be regarded informally as converting the first *non-space* character of the ⟨*tokens*⟩ to uppercase. However, the process is more complex than this as there are some situations where a single lowercase character maps to a special form, for example `ij` in Dutch which becomes `IJ`. There are two functions available for titlecasing: one which applies the change to each "word" and a second which only applies at the start of the input. (Here, "word" boundaries are spaces: at present, full Unicode word breaking is not attempted.)

Importantly, notice that these functions are intended for working with user *text for typesetting*. For case changing programmatic data see the l3str module and discussion there of `\str_lowercase:n`, `\str_uppercase:n` and `\str_casefold:n`.

Case changing does not take place within math mode material so for example

```
\text_uppercase:n { Some~text~$y = mx + c$~with~{Braces} }
```

becomes

```
SOME TEXT $y = mx + c$ WITH {BRACES}
```

The first mandatory argument of commands listed in `\l_text_case_exclude_arg_tl` is excluded from case changing; the latter are entirely non-textual content (such as labels).

The standard mappings here follow those defined by the Unicode Consortium in `UnicodeData.txt` and `SpecialCasing.txt`. For pTeX, only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Locale-sensitive conversions are enabled using the ⟨*BCP-47*⟩ argument, and follow Unicode Consortium guidelines. Currently, the locale strings recognized for special handling are as follows.

- Armenian (`hy` and `hy-x-yiwn`) The setting `hy` maps the codepoint U+0587, the ligature of letters ech and yiwn, to the codepoints for capital ech and vew when uppercasing: this follows the spelling reform which is used in Armenia. The alternative `hy-x-yiwn` maps U+0587 to capital ech and yiwn on uppercasing (also the output if Armenian is not selected at all).

- Azeri and Turkish (`az` and `tr`). The case pairs I/i-dotless and I-dot/i are activated for these languages. The combining dot mark is removed when lowercasing I-dot and introduced when upper casing i-dotless.

- German (`de-x-eszett`). An alternative mapping for German in which the lowercase *Eszett* maps to a *großes Eszett*.

- Greek (`el`). Removes accents from Greek letters when uppercasing; titlecasing leaves accents in place. A variant `el-x-iota` is available which converts the *ypogegrammeni* (subscript muted iota) to capital iota when uppercasing: the standard version retains the subscript versions.
- Lithuanian (`lt`). The lowercase letters i and j should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lowercasing of the relevant uppercase letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when uppercasing in these cases. Note that *only* the accents used in Lithuanian are covered: the behaviour of other accents are not modified.
- Medieval Latin (`la-x-medieval`). The characters `u` and `V` are interchanged on case changing.
- Dutch (`nl`). Capitalisation of `ij` at the beginning of titlecased input produces `IJ` rather than `Ij`.

Determining whether non-letter characters at the start of text should count as the uppercase element is controllable. When `\l_text_titlecase_check_letter_bool` is `true`, codepoints which are not letters (Unicode general category `L`) are not changed, and only the first *letter* is uppercased. When `\l_text_titlecase_check_letter_bool` is `false`, the first codepoint is uppercased, irrespective of the general code of the character.

`\text_declare_case_equivalent:Nn`

New: 2022-07-04

`\text_declare_case_equivalent:Nn` ⟨*cmd*⟩ {⟨*replacement*⟩}

Declares that the ⟨*replacement*⟩ tokens should be used whenever the ⟨*cmd*⟩ (a single token) is encountered during case changing.

`\text_declare_lowercase_mapping:nn`
`\text_declare_lowercase_mapping:nnn`
`\text_declare_titlecase_mapping:nn`
`\text_declare_titlecase_mapping:nnn`
`\text_declare_uppercase_mapping:nn`
`\text_declare_uppercase_mapping:nnn`

New: 2023-04-11
Updated: 2023-04-20

`\text_declare_lowercase_mapping:nn` {⟨*codepoint*⟩} {⟨*replacement*⟩}
`\text_declare_lowercase_mapping:nnn` {⟨*BCP-47*⟩} {⟨*codepoint*⟩} {⟨*replacement*⟩}

Declares that the ⟨*replacement*⟩ tokens should be used when case mapping the ⟨*codepoint*⟩, rather than the standard mapping given in the Unicode data files. The `nnn` version takes a BCP-47 tag, which can be used to specify that the customisation only applies to that locale.

`\text_case_switch:nnnn` ⋆

New: 2022-07-04

`\text_case_switch:nnnn` {⟨*normal*⟩} {⟨*upper*⟩} {⟨*lower*⟩} {⟨*title*⟩}

Context-sensitive function which will expand to one of the ⟨*normal*⟩, ⟨*upper*⟩, ⟨*lower*⟩ or ⟨*title*⟩ tokens depending on the current case changing operation. Outside of case changing, the ⟨*normal*⟩ tokens are produced. Within case changing, the appropriate mapping tokens are inserted.

34.3 Removing formatting from text

`\text_purify:n` ⋆ `\text_purify:n {⟨text⟩}`

New: 2020-03-05
Updated: 2020-05-14

Takes user input ⟨*text*⟩ and expands as described for `\text_expand:n`, then removes all functions from the resulting text. Math mode material (as delimited by pairs given in `\l_text_math_delims_tl` or as the argument to commands listed in `\l_text_math_arg_tl`) is left contained in a pair of `$` delimiters. Non-expandable functions present in the ⟨*text*⟩ must either have a defined equivalent (see `\text_declare_purify_equivalent:Nn`) or will be removed from the result. Implicit tokens are converted to their explicit equivalent.

`\text_declare_purify_equivalent:Nn`
`\text_declare_purify_equivalent:Ne`

`\text_declare_purify_equivalent:Nn ⟨cmd⟩ {⟨replacement⟩}`

New: 2020-03-05

Declares that the ⟨*replacement*⟩ tokens should be used whenever the ⟨*cmd*⟩ (a single token) is encountered. The ⟨*replacement*⟩ tokens should be expandable.

34.4 Control variables

`\l_text_math_arg_tl`

Lists commands present in the ⟨*text*⟩ where the argument of the command should be treated as math mode material. The treatment here is similar to `\l_text_math_delims_tl` but for a command rather than paired delimiters.

`\l_text_math_delims_tl`

Lists pairs of tokens which delimit (in-line) math mode content; such content *may* be excluded from processing.

`\l_text_case_exclude_arg_tl`

Lists commands where the first mandatory argument is excluded from case changing.

`\l_text_expand_exclude_tl`

Lists commands which are excluded from expansion. This protection includes everything up to and including their first braced argument.

`\l_text_titlecase_check_letter_bool`

Controls how the start of titlecasing is handled: when `true`, the first *letter* in text is considered. The standard setting is `true`.

34.5 Mapping to graphemes

Grapheme splitting is implemented using the algorithm described in Unicode Standard Annex #29. This includes support for extended grapheme clusters. Text starting with a line feed or carriage return character will drop this due to standard TeX processing. At present extended pictograms are not supported: these may be added in a future release.

`\text_map_function:nN` ☆

New: 2022-08-04

`\text_map_function:nN {⟨text⟩} ⟨function⟩`

Takes user input ⟨*text*⟩ and expands as described for `\text_expand:n`, then maps over the *graphemes* within the result, passing each grapheme to the ⟨*function*⟩. Broadly a grapheme is a "user perceived character": the Unicode Consortium describe the decomposition of input to graphemes in depth, and the approach used here implements that algorithm. The ⟨*function*⟩ should accept one argument as ⟨*balanced text*⟩: this may be comprise codepoints or may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also `\text_map_inline:nn`.

`\text_map_inline:nn`

New: 2022-08-04

`\text_map_inline:nn {⟨text⟩} {⟨inline function⟩}`

Takes user input ⟨*text*⟩ and expands as described for `\text_expand:n`, then maps over the *graphemes* within the result, passing each grapheme to the ⟨*inline function*⟩. Broadly a grapheme is a "user perceived character": the Unicode Consortium describe the decomposition of input to graphemes in depth, and the approach used here implements that algorithm. The ⟨*inline function*⟩ should consist of code which receives the grapheme as ⟨*balanced text*⟩: this may be comprise codepoints or may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also `\text_map_function:nN`.

`\text_map_break:` ☆
`\text_map_break:n` ☆

New: 2022-08-04

`\text_map_break:`
`\text_map_break:n {⟨code⟩}`

Used to terminate a `\text_map_...` function before all entries in the ⟨*text*⟩ have been processed. This normally takes place within a conditional statement.

Part VI
Typesetting

Chapter 35

The l3box module
Boxes

Box variables contain typeset material that can be inserted on the page or in other boxes. Their contents cannot be converted back to lists of tokens. There are three kinds of box operations: horizontal mode denoted with prefix `\hbox_`, vertical mode with prefix `\vbox_`, and the generic operations working in both modes with prefix `\box_`. For instance, a new box variable containing the words "Hello, world!" (in a horizontal box) can be obtained by the following code.

```
\box_new:N \l_hello_box
\hbox_set:Nn \l_hello_box { Hello, ~ world! }
```

The argument is typeset inside a TeX group so that any variables assigned during the construction of this box restores its value afterwards.

Box variables from l3box are compatible with those of LaTeX 2ε and plain TeX and can be used interchangeably. The l3box commands to construct boxes, such as `\hbox:n` or `\hbox_set:Nn`, are "color-safe", meaning that

```
\hbox:n { \color_select:n { blue } Hello, } ~ world!
```

will result in "Hello," taking the color blue, but "world!" remaining with the prevailing color outside the box.

35.1 Creating and initialising boxes

`\box_new:N`
`\box_new:c`

`\box_new:N ⟨box⟩`

Creates a new ⟨*box*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*box*⟩ is initially void.

`\box_clear:N`
`\box_clear:c`
`\box_gclear:N`
`\box_gclear:c`

`\box_clear:N ⟨box⟩`

Clears the content of the ⟨*box*⟩ by setting the box equal to `\c_empty_box`.

\box_clear_new:N
\box_clear_new:c
\box_gclear_new:N
\box_gclear_new:c

\box_clear_new:N ⟨*box*⟩

Ensures that the ⟨*box*⟩ exists globally by applying \box_new:N if necessary, then applies \box_(g)clear:N to leave the ⟨*box*⟩ empty.

\box_set_eq:NN
\box_set_eq:(cN|Nc|cc)
\box_gset_eq:NN
\box_gset_eq:(cN|Nc|cc)

\box_set_eq:NN ⟨box_1⟩ ⟨box_2⟩

Sets the content of ⟨box_1⟩ equal to that of ⟨box_2⟩.

\box_if_exist_p:N ⋆
\box_if_exist_p:c ⋆
\box_if_exist:N*TF* ⋆
\box_if_exist:c*TF* ⋆

New: 2012-03-03

\box_if_exist_p:N ⟨*box*⟩
\box_if_exist:NTF ⟨*box*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*box*⟩ is currently defined. This does not check that the ⟨*box*⟩ really is a box.

35.2 Using boxes

\box_use:N
\box_use:c

\box_use:N ⟨*box*⟩

Inserts the current content of the ⟨*box*⟩ onto the current list for typesetting. An error is raised if the variable does not exist or if it is invalid.

TEXhackers note: This is the TEX primitive \copy.

\box_move_right:nn
\box_move_left:nn

\box_move_right:nn {⟨*dim expr*⟩} {⟨*box function*⟩}

This function operates in vertical mode, and inserts the material specified by the ⟨*box function*⟩ such that its reference point is displaced horizontally by the given ⟨*dim expr*⟩ from the reference point for typesetting, to the right or left as appropriate. The ⟨*box function*⟩ should be a box operation such as \box_use:N \<box> or a "raw" box specification such as \vbox:n { xyz }.

\box_move_up:nn
\box_move_down:nn

\box_move_up:nn {⟨*dim expr*⟩} {⟨*box function*⟩}

This function operates in horizontal mode, and inserts the material specified by the ⟨*box function*⟩ such that its reference point is displaced vertically by the given ⟨*dim expr*⟩ from the reference point for typesetting, up or down as appropriate. The ⟨*box function*⟩ should be a box operation such as \box_use:N \<box> or a "raw" box specification such as \vbox:n { xyz }.

35.3 Measuring and setting box dimensions

\box_dp:N
\box_dp:c

\box_dp:N ⟨*box*⟩

Calculates the depth (below the baseline) of the ⟨*box*⟩ in a form suitable for use in a ⟨*dim expr*⟩.

TeXhackers note: This is the TeX primitive \dp.

\box_ht:N
\box_ht:c

\box_ht:N ⟨*box*⟩

Calculates the height (above the baseline) of the ⟨*box*⟩ in a form suitable for use in a ⟨*dim expr*⟩.

TeXhackers note: This is the TeX primitive \ht.

\box_wd:N
\box_wd:c

\box_wd:N ⟨*box*⟩

Calculates the width of the ⟨*box*⟩ in a form suitable for use in a ⟨*dim expr*⟩.

TeXhackers note: This is the TeX primitive \wd.

\box_ht_plus_dp:N
\box_ht_plus_dp:c
New: 2021-05-05

\box_ht_plus_dp:N ⟨*box*⟩

Calculates the total vertical size (height plus depth) of the ⟨*box*⟩ in a form suitable for use in a ⟨*dim expr*⟩.

\box_set_dp:Nn
\box_set_dp:cn
\box_gset_dp:Nn
\box_gset_dp:cn
Updated: 2019-01-22

\box_set_dp:Nn ⟨*box*⟩ {⟨*dim expr*⟩}

Set the depth (below the baseline) of the ⟨*box*⟩ to the value of the {⟨*dim expr*⟩}.

\box_set_ht:Nn
\box_set_ht:cn
\box_gset_ht:Nn
\box_gset_ht:cn
Updated: 2019-01-22

\box_set_ht:Nn ⟨*box*⟩ {⟨*dim expr*⟩}

Set the height (above the baseline) of the ⟨*box*⟩ to the value of the {⟨*dim expr*⟩}.

\box_set_wd:Nn
\box_set_wd:cn
\box_gset_wd:Nn
\box_gset_wd:cn
Updated: 2019-01-22

\box_set_wd:Nn ⟨*box*⟩ {⟨*dim expr*⟩}

Set the width of the ⟨*box*⟩ to the value of the {⟨*dim expr*⟩}.

35.4 Box conditionals

\box_if_empty_p:N ⋆
\box_if_empty_p:c ⋆
\box_if_empty:NTF ⋆
\box_if_empty:cTF ⋆

\box_if_empty_p:N ⟨box⟩
\box_if_empty:NTF ⟨box⟩ {⟨true code⟩} {⟨false code⟩}

Tests if ⟨box⟩ is a empty (equal to \c_empty_box).

\box_if_horizontal_p:N ⋆
\box_if_horizontal_p:c ⋆
\box_if_horizontal:NTF ⋆
\box_if_horizontal:cTF ⋆

\box_if_horizontal_p:N ⟨box⟩
\box_if_horizontal:NTF ⟨box⟩ {⟨true code⟩} {⟨false code⟩}

Tests if ⟨box⟩ is a horizontal box.

\box_if_vertical_p:N ⋆
\box_if_vertical_p:c ⋆
\box_if_vertical:NTF ⋆
\box_if_vertical:cTF ⋆

\box_if_vertical_p:N ⟨box⟩
\box_if_vertical:NTF ⟨box⟩ {⟨true code⟩} {⟨false code⟩}

Tests if ⟨box⟩ is a vertical box.

35.5 The last box inserted

\box_set_to_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c

\box_set_to_last:N ⟨box⟩

Sets the ⟨box⟩ equal to the last item (box) added to the current partial list, removing the item from the list at the same time. When applied to the main vertical list, the ⟨box⟩ is always void as it is not possible to recover the last added item.

35.6 Constant boxes

\c_empty_box

Updated: 2012-11-04

This is a permanently empty box, which is neither set as horizontal nor vertical.

TeXhackers note: At the TeX level this is a void box.

35.7 Scratch boxes

\l_tmpa_box
\l_tmpb_box

Updated: 2012-11-04

Scratch boxes for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_box
\g_tmpb_box

Scratch boxes for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

35.8 Viewing box contents

`\box_show:N`
`\box_show:c`

Updated: 2012-05-11

`\box_show:N` ⟨*box*⟩

Shows full details of the content of the ⟨*box*⟩ in the terminal.

`\box_show:Nnn`
`\box_show:cnn`

New: 2012-05-11

`\box_show:Nnn` ⟨*box*⟩ {⟨*int expr*$_1$⟩} {⟨*int expr*$_2$⟩}

Display the contents of ⟨*box*⟩ in the terminal, showing the first ⟨*int expr*$_1$⟩ items of the box, and descending into ⟨*int expr*$_2$⟩ group levels.

`\box_log:N`
`\box_log:c`

New: 2012-05-11

`\box_log:N` ⟨*box*⟩

Writes full details of the content of the ⟨*box*⟩ to the log.

`\box_log:Nnn`
`\box_log:cnn`

New: 2012-05-11

`\box_log:Nnn` ⟨*box*⟩ {⟨*int expr*$_1$⟩} {⟨*int expr*$_2$⟩}

Writes the contents of ⟨*box*⟩ to the log, showing the first ⟨*int expr*$_1$⟩ items of the box, and descending into ⟨*int expr*$_2$⟩ group levels.

35.9 Boxes and color

All LaTeX3 boxes are "color safe": a color set inside the box stops applying after the end of the box has occurred.

35.10 Horizontal mode boxes

`\hbox:n`

Updated: 2017-04-05

`\hbox:n` {⟨*contents*⟩}

Typesets the ⟨*contents*⟩ into a horizontal box of natural width and then includes this box in the current list for typesetting.

`\hbox_to_wd:nn`

Updated: 2017-04-05

`\hbox_to_wd:nn` {⟨*dim expr*⟩} {⟨*contents*⟩}

Typesets the ⟨*contents*⟩ into a horizontal box of width ⟨*dim expr*⟩ and then includes this box in the current list for typesetting.

`\hbox_to_zero:n`

Updated: 2017-04-05

`\hbox_to_zero:n` {⟨*contents*⟩}

Typesets the ⟨*contents*⟩ into a horizontal box of zero width and then includes this box in the current list for typesetting.

`\hbox_set:Nn`
`\hbox_set:cn`
`\hbox_gset:Nn`
`\hbox_gset:cn`

Updated: 2017-04-05

`\hbox_set:Nn` ⟨*box*⟩ {⟨*contents*⟩}

Typesets the ⟨*contents*⟩ at natural width and then stores the result inside the ⟨*box*⟩.

\hbox_set_to_wd:Nnn
\hbox_set_to_wd:cnn
\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cnn

Updated: 2017-04-05

\hbox_set_to_wd:Nnn ⟨*box*⟩ {⟨*dim expr*⟩} {⟨*contents*⟩}

Typesets the ⟨*contents*⟩ to the width given by the ⟨*dim expr*⟩ and then stores the result inside the ⟨*box*⟩.

\hbox_overlap_center:n

New: 2020-08-25

\hbox_overlap_center:n {⟨*contents*⟩}

Typesets the ⟨*contents*⟩ into a horizontal box of zero width such that material protrudes equally to both sides of the insertion point.

\hbox_overlap_right:n

Updated: 2017-04-05

\hbox_overlap_right:n {⟨*contents*⟩}

Typesets the ⟨*contents*⟩ into a horizontal box of zero width such that material protrudes to the right of the insertion point.

\hbox_overlap_left:n

Updated: 2017-04-05

\hbox_overlap_left:n {⟨*contents*⟩}

Typesets the ⟨*contents*⟩ into a horizontal box of zero width such that material protrudes to the left of the insertion point.

\hbox_set:Nw
\hbox_set:cw
\hbox_set_end:
\hbox_gset:Nw
\hbox_gset:cw
\hbox_gset_end:

Updated: 2017-04-05

\hbox_set:Nw ⟨*box*⟩ ⟨*contents*⟩ \hbox_set_end:

Typesets the ⟨*contents*⟩ at natural width and then stores the result inside the ⟨*box*⟩. In contrast to **\hbox_set:Nn** this function does not absorb the argument when finding the ⟨*content*⟩, and so can be used in circumstances where the ⟨*content*⟩ may not be a simple argument.

\hbox_set_to_wd:Nnw
\hbox_set_to_wd:cnw
\hbox_gset_to_wd:Nnw
\hbox_gset_to_wd:cnw

New: 2017-06-08

\hbox_set_to_wd:Nnw ⟨*box*⟩ {⟨*dim expr*⟩} ⟨*contents*⟩ \hbox_set_end:

Typesets the ⟨*contents*⟩ to the width given by the ⟨*dim expr*⟩ and then stores the result inside the ⟨*box*⟩. In contrast to **\hbox_set_to_wd:Nnn** this function does not absorb the argument when finding the ⟨*content*⟩, and so can be used in circumstances where the ⟨*content*⟩ may not be a simple argument

\hbox_unpack:N
\hbox_unpack:c

\hbox_unpack:N ⟨*box*⟩

Unpacks the content of the horizontal ⟨*box*⟩, retaining any stretching or shrinking applied when the ⟨*box*⟩ was set.

TEXhackers note: This is the TEX primitive \unhcopy.

35.11 Vertical mode boxes

Vertical boxes inherit their baseline from their contents. The standard case is that the baseline of the box is at the same position as that of the last item added to the box. This means that the box has no depth unless the last item added to it had depth. As a result most vertical boxes have a large height value and small or zero depth. The exception are `_top` boxes, where the reference point is that of the first item added. These tend to have a large depth and small height, although the latter is typically non-zero.

`\vbox:n`

Updated: 2017-04-05

`\vbox:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a vertical box of natural height and includes this box in the current list for typesetting.

`\vbox_top:n`

Updated: 2017-04-05

`\vbox_top:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box is equal to that of the *first* item added to the box.

`\vbox_to_ht:nn`

Updated: 2017-04-05

`\vbox_to_ht:nn {⟨dim expr⟩} {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a vertical box of height ⟨*dim expr*⟩ and then includes this box in the current list for typesetting.

`\vbox_to_zero:n`

Updated: 2017-04-05

`\vbox_to_zero:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a vertical box of zero height and then includes this box in the current list for typesetting.

`\vbox_set:Nn`
`\vbox_set:cn`
`\vbox_gset:Nn`
`\vbox_gset:cn`

Updated: 2017-04-05

`\vbox_set:Nn ⟨box⟩ {⟨contents⟩}`

Typesets the ⟨*contents*⟩ at natural height and then stores the result inside the ⟨*box*⟩.

`\vbox_set_top:Nn`
`\vbox_set_top:cn`
`\vbox_gset_top:Nn`
`\vbox_gset_top:cn`

Updated: 2017-04-05

`\vbox_set_top:Nn ⟨box⟩ {⟨contents⟩}`

Typesets the ⟨*contents*⟩ at natural height and then stores the result inside the ⟨*box*⟩. The baseline of the box is equal to that of the *first* item added to the box.

`\vbox_set_to_ht:Nnn`
`\vbox_set_to_ht:cnn`
`\vbox_gset_to_ht:Nnn`
`\vbox_gset_to_ht:cnn`

Updated: 2017-04-05

`\vbox_set_to_ht:Nnn ⟨box⟩ {⟨dim expr⟩} {⟨contents⟩}`

Typesets the ⟨*contents*⟩ to the height given by the ⟨*dim expr*⟩ and then stores the result inside the ⟨*box*⟩.

`\vbox_set:Nw`
`\vbox_set:cw`
`\vbox_set_end:`
`\vbox_gset:Nw`
`\vbox_gset:cw`
`\vbox_gset_end:`

Updated: 2017-04-05

`\vbox_set:Nw ⟨box⟩ ⟨contents⟩ \vbox_set_end:`

Typesets the ⟨*contents*⟩ at natural height and then stores the result inside the ⟨*box*⟩. In contrast to `\vbox_set:Nn` this function does not absorb the argument when finding the ⟨*content*⟩, and so can be used in circumstances where the ⟨*content*⟩ may not be a simple argument.

\vbox_set_to_ht:Nnw
\vbox_set_to_ht:cnw
\vbox_gset_to_ht:Nnw
\vbox_gset_to_ht:cnw

New: 2017-06-08

\vbox_set_to_ht:Nnw ⟨box⟩ {⟨dim expr⟩} ⟨contents⟩ \vbox_set_end:

Typesets the ⟨contents⟩ to the height given by the ⟨dim expr⟩ and then stores the result inside the ⟨box⟩. In contrast to \vbox_set_to_ht:Nnn this function does not absorb the argument when finding the ⟨content⟩, and so can be used in circumstances where the ⟨content⟩ may not be a simple argument

\vbox_set_split_to_ht:NNn
\vbox_set_split_to_ht:(cNn|Ncn|ccn)
\vbox_gset_split_to_ht:NNn
\vbox_gset_split_to_ht:(cNn|Ncn|ccn)

Updated: 2018-12-29

\vbox_set_split_to_ht:NNn ⟨box_1⟩ ⟨box_2⟩ {⟨dim expr⟩}

Sets ⟨box_1⟩ to contain material to the height given by the ⟨dim expr⟩ by removing content from the top of ⟨box_2⟩ (which must be a vertical box).

\vbox_unpack:N
\vbox_unpack:c

\vbox_unpack:N ⟨box⟩

Unpacks the content of the vertical ⟨box⟩, retaining any stretching or shrinking applied when the ⟨box⟩ was set.

TeXhackers note: This is the TeX primitive \unvcopy.

35.12 Using boxes efficiently

The functions above for using box contents work in exactly the same way as for any other expl3 variable. However, for efficiency reasons, it is also useful to have functions which *drop* box contents on use. When a box is dropped, the box becomes empty at the group level *where the box was originally set* rather than necessarily *at the current group level*. For example, with

```
\hbox_set:Nn \l_tmpa_box { A }
\group_begin:
  \hbox_set:Nn \l_tmpa_box { B }
   \group_begin:
   \box_use_drop:N \l_tmpa_box
  \group_end:
  \box_show:N \l_tmpa_box
\group_end:
\box_show:N \l_tmpa_box
```

the first use of \box_show:N will show an entirely cleared (void) box, and the second will show the letter A in the box.

These functions should be preferred when the content of the box is no longer required after use. Note that due to the unusual scoping behaviour of drop functions they may be applied to both local and global boxes: the latter will naturally be set and thus cleared at a global level.

\box_use_drop:N
\box_use_drop:c

\box_use_drop:N ⟨*box*⟩

Inserts the current content of the ⟨*box*⟩ onto the current list for typesetting then drops the box content. An error is raised if the variable does not exist or if it is invalid. This function may be applied to local or global boxes.

TEXhackers note: This is the TEX primitive \box.

\box_set_eq_drop:NN
\box_set_eq_drop:(cN|Nc|cc)

New: 2019-01-17

\box_set_eq_drop:NN ⟨*box*$_1$⟩ ⟨*box*$_2$⟩

Sets the content of ⟨*box*$_1$⟩ equal to that of ⟨*box*$_2$⟩, then drops ⟨*box*$_2$⟩.

\box_gset_eq_drop:NN
\box_gset_eq_drop:(cN|Nc|cc)

New: 2019-01-17

\box_gset_eq_drop:NN ⟨*box*$_1$⟩ ⟨*box*$_2$⟩

Sets the content of ⟨*box*$_1$⟩ globally equal to that of ⟨*box*$_2$⟩, then drops ⟨*box*$_2$⟩.

\hbox_unpack_drop:N
\hbox_unpack_drop:c

New: 2019-01-17

\hbox_unpack_drop:N ⟨*box*⟩

Unpacks the content of the horizontal ⟨*box*⟩, retaining any stretching or shrinking applied when the ⟨*box*⟩ was set. The original ⟨*box*⟩ is then dropped.

TEXhackers note: This is the TEX primitive \unhbox.

\vbox_unpack_drop:N
\vbox_unpack_drop:c

New: 2019-01-17

\vbox_unpack_drop:N ⟨*box*⟩

Unpacks the content of the vertical ⟨*box*⟩, retaining any stretching or shrinking applied when the ⟨*box*⟩ was set. The original ⟨*box*⟩ is then dropped.

TEXhackers note: This is the TEX primitive \unvbox.

35.13 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in TEX by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing of boxed material can best be handled by modifying boxes. These transformations are described here.

`\box_autosize_to_wd_and_ht:Nnn`
`\box_autosize_to_wd_and_ht:cnn`
`\box_gautosize_to_wd_and_ht:Nnn`
`\box_gautosize_to_wd_and_ht:cnn`

New: 2017-04-04
Updated: 2019-01-22

`\box_autosize_to_wd_and_ht:Nnn` ⟨*box*⟩ {⟨*x-size*⟩} {⟨*y-size*⟩}

Resizes the ⟨*box*⟩ to fit within the given ⟨*x-size*⟩ (horizontally) and ⟨*y-size*⟩ (vertically); both of the sizes are dimension expressions. The ⟨*y-size*⟩ is the height only: it does not include any depth. The updated ⟨*box*⟩ is an `hbox`, irrespective of the nature of the ⟨*box*⟩ before the resizing is applied. The final size of the ⟨*box*⟩ is the smaller of {⟨*x-size*⟩} and {⟨*y-size*⟩}, *i.e.* the result fits within the dimensions specified. Negative sizes cause the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ is unchanged. Thus a negative ⟨*y-size*⟩ results in the ⟨*box*⟩ having a depth dependent on the height of the original and *vice versa*.

`\box_autosize_to_wd_and_ht_plus_dp:Nnn`
`\box_autosize_to_wd_and_ht_plus_dp:cnn`
`\box_gautosize_to_wd_and_ht_plus_dp:Nnn`
`\box_gautosize_to_wd_and_ht_plus_dp:cnn`

New: 2017-04-04
Updated: 2019-01-22

`\box_autosize_to_wd_and_ht_plus_dp:Nnn` ⟨*box*⟩ {⟨*x-size*⟩} {⟨*y-size*⟩}

Resizes the ⟨*box*⟩ to fit within the given ⟨*x-size*⟩ (horizontally) and ⟨*y-size*⟩ (vertically); both of the sizes are dimension expressions. The ⟨*y-size*⟩ is the total vertical size (height plus depth). The updated ⟨*box*⟩ is an `hbox`, irrespective of the nature of the ⟨*box*⟩ before the resizing is applied. The final size of the ⟨*box*⟩ is the smaller of {⟨*x-size*⟩} and {⟨*y-size*⟩}, *i.e.* the result fits within the dimensions specified. Negative sizes cause the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ is unchanged. Thus a negative ⟨*y-size*⟩ results in the ⟨*box*⟩ having a depth dependent on the height of the original and *vice versa*.

`\box_resize_to_ht:Nn`
`\box_resize_to_ht:cn`
`\box_gresize_to_ht:Nn`
`\box_gresize_to_ht:cn`

Updated: 2019-01-22

`\box_resize_to_ht:Nn` ⟨*box*⟩ {⟨*y-size*⟩}

Resizes the ⟨*box*⟩ to ⟨*y-size*⟩ (vertically), scaling the horizontal size by the same amount; ⟨*y-size*⟩ is a dimension expression. The ⟨*y-size*⟩ is the height only: it does not include any depth. The updated ⟨*box*⟩ is an `hbox`, irrespective of the nature of the ⟨*box*⟩ before the resizing is applied. A negative ⟨*y-size*⟩ causes the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ is unchanged. Thus a negative ⟨*y-size*⟩ results in the ⟨*box*⟩ having a depth dependent on the height of the original and *vice versa*.

\box_resize_to_ht_plus_dp:Nn
\box_resize_to_ht_plus_dp:cn
\box_gresize_to_ht_plus_dp:Nn
\box_gresize_to_ht_plus_dp:cn

Updated: 2019-01-22

\box_resize_to_ht_plus_dp:Nn ⟨box⟩ {⟨y-size⟩}

Resizes the ⟨box⟩ to ⟨y-size⟩ (vertically), scaling the horizontal size by the same amount; ⟨y-size⟩ is a dimension expression. The ⟨y-size⟩ is the total vertical size (height plus depth). The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. A negative ⟨y-size⟩ causes the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative ⟨y-size⟩ results in the ⟨box⟩ having a depth dependent on the height of the original and *vice versa*.

\box_resize_to_wd:Nn
\box_resize_to_wd:cn
\box_gresize_to_wd:Nn
\box_gresize_to_wd:cn

Updated: 2019-01-22

\box_resize_to_wd:Nn ⟨box⟩ {⟨x-size⟩}

Resizes the ⟨box⟩ to ⟨x-size⟩ (horizontally), scaling the vertical size by the same amount; ⟨x-size⟩ is a dimension expression. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. A negative ⟨x-size⟩ causes the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative ⟨x-size⟩ results in the ⟨box⟩ having a depth dependent on the height of the original and *vice versa*.

\box_resize_to_wd_and_ht:Nnn
\box_resize_to_wd_and_ht:cnn
\box_gresize_to_wd_and_ht:Nnn
\box_gresize_to_wd_and_ht:cnn

New: 2014-07-03
Updated: 2019-01-22

\box_resize_to_wd_and_ht:Nnn ⟨box⟩ {⟨x-size⟩} {⟨y-size⟩}

Resizes the ⟨box⟩ to ⟨x-size⟩ (horizontally) and ⟨y-size⟩ (vertically): both of the sizes are dimension expressions. The ⟨y-size⟩ is the height only and does not include any depth. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. Negative sizes cause the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative ⟨y-size⟩ results in the ⟨box⟩ having a depth dependent on the height of the original and *vice versa*.

\box_resize_to_wd_and_ht_plus_dp:Nnn
\box_resize_to_wd_and_ht_plus_dp:cnn
\box_gresize_to_wd_and_ht_plus_dp:Nnn
\box_gresize_to_wd_and_ht_plus_dp:cnn

New: 2017-04-06
Updated: 2019-01-22

\box_resize_to_wd_and_ht_plus_dp:Nnn ⟨box⟩ {⟨x-size⟩} {⟨y-size⟩}

Resizes the ⟨box⟩ to ⟨x-size⟩ (horizontally) and ⟨y-size⟩ (vertically): both of the sizes are dimension expressions. The ⟨y-size⟩ is the total vertical size (height plus depth). The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. Negative sizes cause the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative ⟨y-size⟩ results in the ⟨box⟩ having a depth dependent on the height of the original and *vice versa*.

\box_rotate:Nn
\box_rotate:cn
\box_grotate:Nn
\box_grotate:cn

Updated: 2019-01-22

\box_rotate:Nn ⟨box⟩ {⟨angle⟩}

Rotates the ⟨box⟩ by ⟨angle⟩ (a ⟨fp expr⟩ in degrees) anti-clockwise about its reference point. The reference point of the updated box is moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the rotation is applied.

\box_scale:Nnn
\box_scale:cnn
\box_gscale:Nnn
\box_gscale:cnn

Updated: 2019-01-22

\box_scale:Nnn ⟨box⟩ {⟨x-scale⟩} {⟨y-scale⟩}

Scales the ⟨box⟩ by factors ⟨x-scale⟩ and ⟨y-scale⟩ in the horizontal and vertical directions, respectively (both scales are ⟨fp expr⟩). The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the scaling is applied. Negative scalings cause the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ is unchanged. Thus a negative ⟨y-scale⟩ results in the ⟨box⟩ having a depth dependent on the height of the original and *vice versa*.

35.14 Viewing part of a box

\box_set_clipped:N
\box_set_clipped:c
\box_gset_clipped:N
\box_gset_clipped:c

Updated: 2023-04-14

\box_set_clipped:N ⟨box⟩

Clips the ⟨box⟩ in the output so that only material inside the bounding box is displayed in the output. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the clipping is applied. Additional box levels are also generated by this operation.

TEXhackers note: Clipping is implemented by the driver, and as such the full content of the box is placed in the output file. Thus clipping does not remove any information from the raw output, and hidden material can therefore be viewed by direct examination of the file.

\box_set_trim:Nnnnn
\box_set_trim:cnnnn
\box_gset_trim:Nnnnn
\box_gset_trim:cnnnn

New: 2019-01-23

\box_set_trim:Nnnnn ⟨box⟩ {⟨left⟩} {⟨bottom⟩} {⟨right⟩} {⟨top⟩}

Adjusts the bounding box of the ⟨box⟩: ⟨left⟩ is removed from the left-hand edge of the bounding box, ⟨right⟩ from the right-hand edge, and so forth. All adjustments are ⟨dim exprs⟩. Material outside of the bounding box is still displayed in the output unless \box_set_clipped:N is subsequently applied. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the trim operation is applied. Additional box levels are also generated by this operation. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

\box_set_viewport:Nnnnn
\box_set_viewport:cnnnn
\box_gset_viewport:Nnnnn
\box_gset_viewport:cnnnn

New: 2019-01-23

\box_set_viewport:Nnnnn ⟨box⟩ {⟨llx⟩} {⟨lly⟩} {⟨urx⟩} {⟨ury⟩}

Adjusts the bounding box of the ⟨box⟩ such that it has lower-left coordinates (⟨llx⟩, ⟨lly⟩) and upper-right coordinates (⟨urx⟩, ⟨ury⟩). All four coordinate positions are ⟨dim exprs⟩. Material outside of the bounding box is still displayed in the output unless \box_set_clipped:N is subsequently applied. The updated ⟨box⟩ is an hbox, irrespective of the nature of the ⟨box⟩ before the viewport operation is applied. Additional box levels are also generated by this operation.

35.15 Primitive box conditionals

`\if_hbox:N ⋆`

```
\if_hbox:N ⟨box⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Tests is ⟨*box*⟩ is a horizontal box.

TeXhackers note: This is the TeX primitive `\ifhbox`.

`\if_vbox:N ⋆`

```
\if_vbox:N ⟨box⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Tests is ⟨*box*⟩ is a vertical box.

TeXhackers note: This is the TeX primitive `\ifvbox`.

`\if_box_empty:N ⋆`

```
\if_box_empty:N ⟨box⟩
  ⟨true code⟩
\else:
  ⟨false code⟩
\fi:
```

Tests is ⟨*box*⟩ is an empty (void) box.

TeXhackers note: This is the TeX primitive `\ifvoid`.

Chapter 36

The l3coffins module Coffin code layer

The material in this module provides the low-level support system for coffins. For details about the design concept of a coffin, see the xcoffins module (in the l3experimental bundle).

36.1 Creating and initialising coffins

\coffin_new:N
\coffin_new:c

New: 2011-08-17

\coffin_new:N ⟨coffin⟩

Creates a new ⟨coffin⟩ or raises an error if the name is already taken. The declaration is global. The ⟨coffin⟩ is initially empty.

\coffin_clear:N
\coffin_clear:c
\coffin_gclear:N
\coffin_gclear:c

New: 2011-08-17
Updated: 2019-01-21

\coffin_clear:N ⟨coffin⟩

Clears the content of the ⟨coffin⟩.

\coffin_set_eq:NN
\coffin_set_eq:(Nc|cN|cc)
\coffin_gset_eq:NN
\coffin_gset_eq:(Nc|cN|cc)

New: 2011-08-17
Updated: 2019-01-21

\coffin_set_eq:NN ⟨coffin$_1$⟩ ⟨coffin$_2$⟩

Sets both the content and poles of ⟨coffin$_1$⟩ equal to those of ⟨coffin$_2$⟩.

\coffin_if_exist_p:N ⋆
\coffin_if_exist_p:c ⋆
\coffin_if_exist:NTF ⋆
\coffin_if_exist:cTF ⋆

New: 2012-06-20

\coffin_if_exist_p:N ⟨coffin⟩
\coffin_if_exist:NTF ⟨coffin⟩ {⟨true code⟩} {⟨false code⟩}

Tests whether the ⟨coffin⟩ is currently defined.

36.2 Setting coffin content and poles

\hcoffin_set:Nn
\hcoffin_set:cn
\hcoffin_gset:Nn
\hcoffin_gset:cn

New: 2011-08-17
Updated: 2019-01-21

\hcoffin_set:Nn ⟨*coffin*⟩ {⟨*material*⟩}

Typesets the ⟨*material*⟩ in horizontal mode, storing the result in the ⟨*coffin*⟩. The standard poles for the ⟨*coffin*⟩ are then set up based on the size of the typeset material.

\hcoffin_set:Nw
\hcoffin_set:cw
\hcoffin_set_end:
\hcoffin_gset:Nw
\hcoffin_gset:cw
\hcoffin_gset_end:

New: 2011-09-10
Updated: 2019-01-21

\hcoffin_set:Nw ⟨*coffin*⟩ ⟨*material*⟩ \hcoffin_set_end:

Typesets the ⟨*material*⟩ in horizontal mode, storing the result in the ⟨*coffin*⟩. The standard poles for the ⟨*coffin*⟩ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

\vcoffin_set:Nnn
\vcoffin_set:cnn
\vcoffin_gset:Nnn
\vcoffin_gset:cnn

New: 2011-08-17
Updated: 2023-02-03

\vcoffin_set:Nnn ⟨*coffin*⟩ {⟨*width*⟩} {⟨*material*⟩}

Typesets the ⟨*material*⟩ in vertical mode constrained to the given ⟨*width*⟩ and stores the result in the ⟨*coffin*⟩. The standard poles for the ⟨*coffin*⟩ are then set up based on the size of the typeset material.

\vcoffin_set:Nnw
\vcoffin_set:cnw
\vcoffin_set_end:
\vcoffin_gset:Nnw
\vcoffin_gset:cnw
\vcoffin_gset_end:

New: 2011-09-10
Updated: 2023-02-03

\vcoffin_set:Nnw ⟨*coffin*⟩ {⟨*width*⟩} ⟨*material*⟩ \vcoffin_set_end:

Typesets the ⟨*material*⟩ in vertical mode constrained to the given ⟨*width*⟩ and stores the result in the ⟨*coffin*⟩. The standard poles for the ⟨*coffin*⟩ are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

\coffin_set_horizontal_pole:Nnn
\coffin_set_horizontal_pole:cnn
\coffin_gset_horizontal_pole:Nnn
\coffin_gset_horizontal_pole:cnn

New: 2012-07-20
Updated: 2019-01-21

\coffin_set_horizontal_pole:Nnn ⟨*coffin*⟩
  {⟨*pole*⟩} {⟨*offset*⟩}

Sets the ⟨*pole*⟩ to run horizontally through the ⟨*coffin*⟩. The ⟨*pole*⟩ is placed at the ⟨*offset*⟩ from the baseline of the ⟨*coffin*⟩. The ⟨*offset*⟩ should be given as a dimension expression.

\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnn
\coffin_gset_vertical_pole:Nnn
\coffin_gset_vertical_pole:cnn

New: 2012-07-20
Updated: 2019-01-21

\coffin_set_vertical_pole:Nnn ⟨*coffin*⟩ {⟨*pole*⟩} {⟨*offset*⟩}

Sets the ⟨*pole*⟩ to run vertically through the ⟨*coffin*⟩. The ⟨*pole*⟩ is placed at the ⟨*offset*⟩ from the left-hand edge of the bounding box of the ⟨*coffin*⟩. The ⟨*offset*⟩ should be given as a dimension expression.

\coffin_reset_poles:N
\coffin_greset_poles:N

New: 2023-05-17

\coffin_reset_poles:N ⟨*coffin*⟩

Resets the poles of the ⟨*coffin*⟩ to the standard set, removing any custom or inherited poles. The poles will therefore be equal to those that would be obtained from `\hcoffin_set:Nn` or similar; the bounding box of the coffin is not reset, so any material outside of the formal bounding box will not influence the poles.

36.3 Coffin affine transformations

\coffin_resize:Nnn
\coffin_resize:cnn
\coffin_gresize:Nnn
\coffin_gresize:cnn

Updated: 2019-01-23

\coffin_resize:Nnn ⟨*coffin*⟩ {⟨*width*⟩} {⟨*total-height*⟩}

Resized the ⟨*coffin*⟩ to ⟨*width*⟩ and ⟨*total-height*⟩, both of which should be given as dimension expressions.

\coffin_rotate:Nn
\coffin_rotate:cn
\coffin_grotate:Nn
\coffin_grotate:cn

\coffin_rotate:Nn ⟨*coffin*⟩ {⟨*angle*⟩}

Rotates the ⟨*coffin*⟩ by the given ⟨*angle*⟩ (given in degrees counter-clockwise). This process rotates both the coffin content and poles. Multiple rotations do not result in the bounding box of the coffin growing unnecessarily.

\coffin_scale:Nnn
\coffin_scale:cnn
\coffin_gscale:Nnn
\coffin_gscale:cnn

Updated: 2019-01-23

\coffin_scale:Nnn ⟨*coffin*⟩ {⟨*x-scale*⟩} {⟨*y-scale*⟩}

Scales the ⟨*coffin*⟩ by a factors ⟨*x-scale*⟩ and ⟨*y-scale*⟩ in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.

36.4 Joining and using coffins

\coffin_attach:NnnNnnnn
\coffin_attach:(cnnNnnnn|Nnncnnnn|cnncnnnn)
\coffin_gattach:NnnNnnnn
\coffin_gattach:(cnnNnnnn|Nnncnnnn|cnncnnnn)

Updated: 2019-01-22

```
\coffin_attach:NnnNnnnn
  ⟨coffin1⟩ {⟨coffin1-pole1⟩} {⟨coffin1-pole2⟩}
  ⟨coffin2⟩ {⟨coffin2-pole1⟩} {⟨coffin2-pole2⟩}
  {⟨x-offset⟩} {⟨y-offset⟩}
```

This function attaches ⟨*coffin*$_2$⟩ to ⟨*coffin*$_1$⟩ such that the bounding box of ⟨*coffin*$_1$⟩ is not altered, *i.e.* ⟨*coffin*$_2$⟩ can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating ⟨*handle*$_1$⟩, the point of intersection of ⟨*coffin*$_1$*-pole*$_1$⟩ and ⟨*coffin*$_1$*-pole*$_2$⟩, and ⟨*handle*$_2$⟩, the point of intersection of ⟨*coffin*$_2$*-pole*$_1$⟩ and ⟨*coffin*$_2$*-pole*$_2$⟩. ⟨*coffin*$_2$⟩ is then attached to ⟨*coffin*$_1$⟩ such that the relationship between ⟨*handle*$_1$⟩ and ⟨*handle*$_2$⟩ is described by the ⟨*x-offset*⟩ and ⟨*y-offset*⟩. The two offsets should be given as dimension expressions.

\coffin_join:NnnNnnnn
\coffin_join:(cnnNnnnn|Nnncnnnn|cnncnnnn)
\coffin_gjoin:NnnNnnnn
\coffin_gjoin:(cnnNnnnn|Nnncnnnn|cnncnnnn)

Updated: 2019-01-22

```
\coffin_join:NnnNnnnn
  ⟨coffin1⟩ {⟨coffin1-pole1⟩} {⟨coffin1-pole2⟩}
  ⟨coffin2⟩ {⟨coffin2-pole1⟩} {⟨coffin2-pole2⟩}
  {⟨x-offset⟩} {⟨y-offset⟩}
```

This function joins ⟨*coffin*$_2$⟩ to ⟨*coffin*$_1$⟩ such that the bounding box of ⟨*coffin*$_1$⟩ may expand. The new bounding box covers the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating ⟨*handle*$_1$⟩, the point of intersection of ⟨*coffin*$_1$*-pole*$_1$⟩ and ⟨*coffin*$_1$*-pole*$_2$⟩, and ⟨*handle*$_2$⟩, the point of intersection of ⟨*coffin*$_2$*-pole*$_1$⟩ and ⟨*coffin*$_2$*-pole*$_2$⟩. ⟨*coffin*$_2$⟩ is then attached to ⟨*coffin*$_1$⟩ such that the relationship between ⟨*handle*$_1$⟩ and ⟨*handle*$_2$⟩ is described by the ⟨*x-offset*⟩ and ⟨*y-offset*⟩. The two offsets should be given as dimension expressions.

\coffin_typeset:Nnnnn
\coffin_typeset:cnnnn

Updated: 2012-07-20

```
\coffin_typeset:Nnnnn ⟨coffin⟩ {⟨pole1⟩} {⟨pole2⟩}
  {⟨x-offset⟩} {⟨y-offset⟩}
```

Typesetting is carried out by first calculating ⟨*handle*⟩, the point of intersection of ⟨*pole*$_1$⟩ and ⟨*pole*$_2$⟩. The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the ⟨*handle*⟩ is described by the ⟨*x-offset*⟩ and ⟨*y-offset*⟩. The two offsets should be given as dimension expressions. Typesetting a coffin is therefore analogous to carrying out an alignment where the "parent" coffin is the current insertion point.

36.5 Measuring coffins

\coffin_dp:N
\coffin_dp:c

```
\coffin_dp:N ⟨coffin⟩
```

Calculates the depth (below the baseline) of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dim expr*⟩.

\coffin_ht:N
\coffin_ht:c

\coffin_ht:N ⟨*coffin*⟩

Calculates the height (above the baseline) of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dim expr*⟩.

\coffin_ht_plus_dp:N
\coffin_ht_plus_dp:c
New: 2024-10-01

\coffin_ht_plus_dp:N ⟨*coffin*⟩

Calculates the total vertical size (height plus depth) of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dim expr*⟩.

\coffin_wd:N
\coffin_wd:c

\coffin_wd:N ⟨*coffin*⟩

Calculates the width of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dim expr*⟩.

36.6 Coffin diagnostics

\coffin_display_handles:Nn
\coffin_display_handles:cn
Updated: 2011-09-02

\coffin_display_handles:Nn ⟨*coffin*⟩ {⟨*color*⟩}

This function first calculates the intersections between all of the ⟨*poles*⟩ of the ⟨*coffin*⟩ to give a set of ⟨*handles*⟩. It then prints the ⟨*coffin*⟩ at the current location in the source, with the position of the ⟨*handles*⟩ marked on the coffin. The ⟨*handles*⟩ are labelled as part of this process: the locations of the ⟨*handles*⟩ and the labels are both printed in the ⟨*color*⟩ specified.

\coffin_mark_handle:Nnnn
\coffin_mark_handle:cnnn
Updated: 2011-09-02

\coffin_mark_handle:Nnnn ⟨*coffin*⟩ {⟨*pole*$_1$⟩} {⟨*pole*$_2$⟩} {⟨*color*⟩}

This function first calculates the ⟨*handle*⟩ for the ⟨*coffin*⟩ as defined by the intersection of ⟨*pole*$_1$⟩ and ⟨*pole*$_2$⟩. It then marks the position of the ⟨*handle*⟩ on the ⟨*coffin*⟩. The ⟨*handle*⟩ are labelled as part of this process: the location of the ⟨*handle*⟩ and the label are both printed in the ⟨*color*⟩ specified.

\coffin_show_structure:N
\coffin_show_structure:c
Updated: 2015-08-01

\coffin_show_structure:N ⟨*coffin*⟩

This function shows the structural information about the ⟨*coffin*⟩ in the terminal. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin.

Notice that the poles of a coffin are defined by four values: the x and y coordinates of a point that the pole passes through and the x- and y-components of a vector denoting the direction of the pole. It is the ratio between the later, rather than the absolute values, which determines the direction of the pole.

\coffin_log_structure:N
\coffin_log_structure:c
New: 2014-08-22
Updated: 2015-08-01

\coffin_log_structure:N ⟨*coffin*⟩

This function writes the structural information about the ⟨*coffin*⟩ in the log file. See also \coffin_show_structure:N which displays the result in the terminal.

\coffin_show:N
\coffin_show:c
\coffin_log:N
\coffin_log:c
New: 2021-05-11

\coffin_show:N ⟨*coffin*⟩
\coffin_log:N ⟨*coffin*⟩

Shows full details of poles and contents of the ⟨*coffin*⟩ in the terminal or log file. See \coffin_show_structure:N and \box_show:N to show separately the pole structure and the contents.

\coffin_show:Nnn
\coffin_show:cnn
\coffin_log:Nnn
\coffin_log:cnn

New: 2021-05-11

\coffin_show:Nnn ⟨*coffin*⟩ {⟨*int expr*$_1$⟩} {⟨*int expr*$_2$⟩}
\coffin_log:Nnn ⟨*coffin*⟩ {⟨*int expr*$_1$⟩} {⟨*int expr*$_2$⟩}

Shows poles and contents of the ⟨*coffin*⟩ in the terminal or log file, showing the first ⟨*int expr*$_1$⟩ items in the coffin, and descending into ⟨*int expr*$_2$⟩ group levels. See \coffin_show_structure:N and \box_show:Nnn to show separately the pole structure and the contents.

36.7 Constants and variables

\c_empty_coffin

A permanently empty coffin.

\l_tmpa_coffin
\l_tmpb_coffin

New: 2012-06-19

Scratch coffins for local assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

\g_tmpa_coffin
\g_tmpb_coffin

New: 2019-01-24

Scratch coffins for global assignment. These are never used by the kernel code, and so are safe for use with any LaTeX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

Chapter 37

The l3color module
Color support

37.1 Color in boxes

Controlling the color of text in boxes requires a small number of control functions, so that the boxed material uses the color at the point where it is set, rather than where it is used.

`\color_group_begin:`
`\color_group_end:`

New: 2011-09-03

```
\color_group_begin:
...
\color_group_end:
```

Creates a color group: one used to "trap" color settings. This grouping is built in to for example `\hbox_set:Nn`.

`\color_ensure_current:`

New: 2011-09-03

`\color_ensure_current:`

Ensures that material inside a box uses the foreground color at the point where the box is set, rather than that in force when the box is used. This function should usually be used within a `\color_group_begin:` ... `\color_group_end:` group.

37.2 Color models

A color *model* is a way to represent sets of colors. Different models are particularly suitable for different output methods, *e.g.* screen or print. Parameter-based models can describe a very large number of unique colors, and have a varying number of *axes* which define a color space. In contrast, various proprietary models are available which define *spot* colors (more formally separations).

Core models are used to pass color information to output; these are "native" to l3color. Core models use real numbers in the range $[0, 1]$ to represent values. The core models supported here are

- `gray` Grayscale color, with a single axis running from 0 (fully black) to 1 (fully white)
- `rgb` Red-green-blue color, with three axes, one for each of the components

- `cmyk` Cyan-magenta-yellow-black color, with four axes, one for each of the components

There are also interface models: these are convenient for users but have to be manipulated before storing/passing to the backend. Interface models are primarily integer-based: see below for more detail. The supported interface models are

- `Gray` Grayscale color, with a single axis running from 0 (fully black) to 15 (fully white)
- `hsb` Hue-saturation-brightness color, with three axes, all real values in the range $[0, 1]$ for hue saturation and brightness
- `Hsb` Hue-saturation-brightness color, with three axes, integer in the range $[0, 360]$ for hue, real values in the range $[0, 1]$ for saturation and brightness
- `HSB` Hue-saturation-brightness color, with three axes, integers in the range $[0, 240]$ for hue, saturation and brightness
- `HTML` HTML format representation of RGB color given as a single six-digit hexadecimal number
- `RGB` Red-green-blue color, with three axes, one for each of the components, values as integers from 0 to 255
- `wave` Light wavelength, a real number in the range 380 to 780 (nanometres)

All interface models are internally stored as `rgb`.

Finally, there are a small number of models which are parsed to allow data transfer from xcolor but which should not be used by end-users. These are

- `cmy` Cyan-magenta-yellow color with three axes, one for each of the components; converted to `cmyk`
- `tHsb` "Tuned" hue-saturation-brightness color with three axes, integer in the range $[0, 360]$ for hue, real values in the range $[0, 1]$ for saturation and brightness; converted to `rgb` using the standard tuning map defined by xcolor
- `&spot` Spot color tint with one value; treated as a gray tint as spot color data is not available for extraction

To allow parsing of data from xcolor, any leading model up the first `:` will be discarded; the approach of selecting an internal form for data is *not* used in l3color.

Additional models may be created to allow mixing of separation colors with each other or with those from other models. See Section 37.9 for more detail of color support for additional models.

When color is selected by model, the ⟨*values*⟩ given are specified as a comma-separated list. The length of the list will therefore be determined by the detail of the model involved.

Color models (and interconversion) are complex, and more details are given in the manual to the LaTeX 2ε xcolor package and in the *PostScript Language Reference Manual*, published by Addison–Wesley.

37.3 Color expressions

In addition to allowing specification of color by model and values, l3color also supports color expressions. These are created by combining one or more color names, with the amount of each specified as a value in the range 0–100. The value should be given between `!` symbols in the expression. Thus for example

```
red!50!green
```

is a mixture of 50 % red and 50 % green. A trailing value is interpreted as implicitly followed by `!white`, and so

```
red!25
```

specifies 25 % red mixed with 75 % white.

Where the models for the mixed colors are different, the model of the first color is used. Thus

```
red!50!cyan
```

will result in a color specification using the `rgb` model, made up of 50 % red and 50 % of cyan *expressed in* `rgb`. This may be important as color model interconversion is not exact.

The one exception to the above is where the first model in an expression is `gray`. In this case, the order of mixing is "swapped" internally, so that for example

```
black!50!red
```

has the same result as

```
red!50!black
```

(the predefined colors `black` and `white` use the `gray` model).

Where more than two colors are mixed in an expression, evaluation takes place in a stepwise fashion. Thus in

```
cyan!50!magenta!10!yellow
```

the sub-expression

```
cyan!50!magenta
```

is first evaluated to give an intermediate color specification, before the second step

```
<intermediate>!10!yellow
```

where `<intermediate>` represents this transitory calculated value.

Within a color expression, `.` may be used to represent the color active for typesetting (the current color). This allows for example

```
.!50
```

to mean a mixture of 50 % of current color with white.

(Color expressions supported here are a subset of those provided by the LaTeX 2ε xcolor package. At present, only such features as are clearly useful have been added here.)

37.4 Named colors

Color names are stored in a single namespace, which makes them accessible as part of color expressions. Whilst they are not reserved in a technical sense, the names `black`, `white`, `red`, `green`, `blue`, `cyan`, `magenta` and `yellow` have special meaning and should not be redefined. Color names should be made up of letters, numbers and spaces only: other characters are reserved for use in color expressions. In particular, `.` represents the current color at the start of a color expression.

`\color_set:nn`

`\color_set:nn {⟨name⟩} {⟨color expression⟩}`

Evaluates the ⟨*color expression*⟩ and stores the resulting color specification as the ⟨*name*⟩.

`\color_set:nnn`

`\color_set:nnn {⟨name⟩} {⟨model(s)⟩} {⟨value(s)⟩}`

Stores the color specification equivalent to the ⟨*model(s)*⟩ and ⟨*values*⟩ as the ⟨*name*⟩.

`\color_set_eq:nn`

`\color_set_eq:nn {⟨name1⟩} {⟨name2⟩}`

Copies the color specification in ⟨*name2*⟩ to ⟨*name1*⟩. The special name `.` may be used to represent the current color, allowing it to be saved to a name.

`\color_if_exist_p:n` ⋆
`\color_if_exist:nTF` ⋆

New: 2022-08-12

`\color_if_exist_p:n {⟨name⟩}`
`\color_if_exist:nTF {⟨name⟩} {⟨true code⟩} {⟨false code⟩}`

Tests whether ⟨*name*⟩ is currently defined to provide a color specification.

`\color_show:n`
`\color_log:n`

New: 2021-05-11

`\color_show:n {⟨name⟩}`
`\color_log:n {⟨name⟩}`

Displays the color specification stored in the ⟨*name*⟩ on the terminal or log file.

37.5 Selecting colors

General selection of color is safe when split across pages: a stack is used to ensure that the correct color is re-selected on the new page.

These commands set the current color (`.`): other more specialised functions such as fill and stroke selectors do *not* adjust this value.

`\color_select:n`

`\color_select:n {⟨color expression⟩}`

Parses the ⟨*color expression*⟩ and then activates the resulting color specification for typeset material.

`\color_select:nn`

`\color_select:nn {⟨model(s)⟩} {⟨value(s)⟩}`

Activates the color specification equivalent to the ⟨*model(s)*⟩ and ⟨*value(s)*⟩ for typeset material.

`\l_color_fixed_model_tl`

When this is set to a non-empty value, colors will be converted to the specified model when they are selected. Note that included images and similar are not influenced by this setting.

37.6 Colors for fills and strokes

Colors for drawing operations and so forth are split into strokes and fills (the latter may also be referred to as non-stroke color). The fill color is used for text under normal circumstances. Depending on the backend, stroke color may use a *stack*, in which case it exhibits the same page breaking behavior as general color. However, `dvips`/`dvisvgm` do not support this, and so color will need to be contained within a scope, such as `\draw_begin:`/`\draw_end:`.

`\color_fill:n`
`\color_stroke:n`

`\color_fill:n {⟨color expression⟩}`

Parses the ⟨*color expression*⟩ and then activates the resulting color specification for filling or stroking.

`\color_fill:nn`
`\color_stroke:nn`

`\color_fill:nn {⟨model(s)⟩} {⟨value(s)⟩}`

Activates the color specification equivalent to the ⟨*model(s)*⟩ and ⟨*value(s)*⟩ for filling or stroking.

`color.sc` When using `dvips`, this PostScript variable holds the stroke color.

37.6.1 Coloring math mode material

Coloring math mode material using `\color_select:nn(n)` has some restrictions and often leads to spacing issues and/or poor input syntax. Avoiding generating `\mathord` atoms whilst coloring only those parts of the input which are required needs careful handling. The functionality here covers this important use case.

`\color_math:nn`
`\color_math:nnn`
New: 2022-01-26

`\color_math:nn {⟨color expression⟩} {⟨content⟩}`
`\color_math:nnn {⟨model(s)⟩} {⟨value(s)⟩} {⟨content⟩}`

Works as for `\color_select:n(n)` but applies color only to the math mode ⟨*content*⟩. The function does not generate a group and the ⟨*content*⟩ therefore retains its math atom states. Sub/superscripts are also properly handled.

`\l_color_math_active_tl`
New: 2022-01-26

This list controls which tokens are considered as math active and should therefore be replaced by their definition during searching for sub/superscripts.

37.7 Multiple color models

When selecting or setting a color with an explicit model, it is possible to give values for more than one model at one time. This is particularly useful where automated conversion between models does not give the desired outcome. To do this, the list of models and list of values are both subdivided using `/` characters (as for the similar function in xcolor). For example, to save a color with explicit `cmyk` and `rgb` values, one could use

```
\color_set:nnn { foo } { cmyk / rgb }
  { 0.1 , 0.2 , 0.3 , 0.4 / 0.1, 0.2 , 0.3 }
```

The manually-specified conversion will be used in preference to automated calculation whenever the model(s) listed are used: both in expressions and when a fixed model is active.

Similarly, the same syntax can be applied to directly selecting a color.

```
\color_select:nn { cmyk / rgb }
  { 0.1 , 0.2 , 0.3 , 0.4 / 0.1, 0.2 , 0.3 }
```

Again, this list is used when a fixed model is active: the first entry is used unless there is a fixed model matching one of the other entries.

37.8 Exporting color specifications

The major use of color expressions is in setting typesetting output, but there are other places in which some form of color information is required. These may need data in a different format or using a different model to the internal representation. Thus a set of functions are available to export colors in different formats.

Valid export targets are

- `backend` Two brace groups: the first containing the model, the second containing space-separated values appropriate for the model; this is the format required by backend functions of expl3
- `comma-sep-cmyk` Comma-separated cyan-magenta-yellow-black values
- `comma-sep-rgb` Comma-separated red-green-blue values suitable for use as a PDF annotation color
- `HTML` Uppercase two-digit hexadecimal values, expressing a red-green-blue color; the digits are *not* separated
- `space-sep-cmyk` Space-separated cyan-magenta-yellow-black values
- `space-sep-rgb` Space-separated red-green-blue values suitable for use as a PDF annotation color

`\color_export:nnN` `\color_export:nnN {⟨color expression⟩} {⟨format⟩} ⟨tl var⟩`

Parses the ⟨*color expression*⟩ as described earlier, then converts to the ⟨*format*⟩ specified and assigns the data to the ⟨*tl var*⟩.

`\color_export:nnnN` `\color_export:nnnN {⟨model⟩} {⟨value(s)⟩} {⟨format⟩} ⟨tl var⟩`

Expresses the combination of ⟨*model*⟩ and ⟨*value(s)*⟩ in an internal representation, then converts to the ⟨*format*⟩ specified and assigns the data to the ⟨*tl var*⟩.

37.9 Creating new color models

Additional color models are required to support specialist workflows, for example those involving separations (see https://helpx.adobe.com/indesign/using/spot-process-colors.html for details of the use of separations in print). Color models may be split into families; for the standard device-based color models (`DeviceCMYK`, `DeviceRGB`, `DeviceGray`), these are synonymous. This is not generally the case: see the PDF reference for more details. (Note that l3color uses the shorter names `cmyk`, etc.)

`\color_model_new:nnn`

`\color_model_new:nnn {⟨model⟩} {⟨family⟩} {⟨params⟩}`

Creates a new ⟨*model*⟩ which is derived from the color model ⟨*family*⟩. The latter should be one of

- `DeviceN`
- `ICCBased`
- `Separation`

(The ⟨*family*⟩ may be given in mixed case as-in the PDF reference: internally, case of these strings is folded.) Depending on the ⟨*family*⟩, one or more ⟨*params*⟩ are mandatory or optional.

For a `Separation` space, there are three *compulsory* keys.

- `name` The name of the Separation, for example the formal name of a spot color ink. Such a ⟨*name*⟩ may contain spaces, etc., which are not permitted in the ⟨*model*⟩.
- `alternative-model` An alternative device colorspace, one of `cmyk`, `rgb`, `gray` or `CIELAB`. The three parameter-based models work as described above; see below for details of CIELAB colors.
- `alternative-values` A comma-separated list of values appropriate to the `alternative-model`. This information is used by the PDF application if the `Separation` is not available.

CIELAB color separations are created using the `alternative-model = CIELAB` setting. These colors must also have an `illuminant` key, one of `a`, `c`, `e`, `d50`, `d55`, `d65` or `d75`. The `alternative-values` in this case are the three parameters $L*$, $a*$ and $b*$ of the CIELAB model. Full details of this device-independent color approach are given in the documentation to the colorspace package.

CIELAB colors *cannot* be converted into other device-dependent color spaces, and as such, mixing can only occur if colors set up using the CIELAB model are also given with an alternative parameter-based model. If that is not the case, l3color will fallback to using black as the colorant in any mixing.

For a `DeviceN` space, there is one *compulsory* key.

- `names` The names of the components of the `DeviceN` space. Each should be either the ⟨*name*⟩ of a `Separation` model, a process color name (`cyan`, etc.) or the special name `none`.

For a `ICCBased` space, there is one *compulsory* key.

- `file` The name of the file containing the profile.

37.9.1 Color profiles

Color profiles are used to ensure color accuracy by linking to collaboration. Applying a profile can be used to standardise color which is otherwise device-dependent.

`\color_profile_apply:nn`

New: 2021-02-23

`\color_profile_apply:nn {⟨profile⟩} {⟨model⟩}`

This function applies a ⟨*profile*⟩ to one of the device ⟨*models*⟩. The profile will then apply to all color of the selected ⟨*model*⟩. The ⟨*profile*⟩ should specify an ICC profile file. The ⟨*model*⟩ has to be one the standard device models: `cmyk`, `gray` or `rgb`.

Chapter 38

The l3pdf module
Core PDF support

38.1 Objects

38.1.1 Named objects

An ⟨*object*⟩ name should fully expand to tokens suitable for use in a label-like context.

`\pdf_object_new:n`
New: 2022-08-23

`\pdf_object_new:n {⟨object⟩}`

Declares ⟨*object*⟩ as a PDF object. The object may be referenced from this point on, and written later using `\pdf_object_write:nnn`.

`\pdf_object_write:nnn`
`\pdf_object_write:nne`
New: 2022-08-23

`\pdf_object_write:nnn {⟨object⟩} {⟨type⟩} {⟨content⟩}`

Writes the ⟨*content*⟩ as content of the ⟨*object*⟩. Depending on the ⟨*type*⟩ declared for the object, the format required for ⟨*content*⟩ will vary:

`array` A space-separated list of values

`dict` Key–value pairs in the form `/⟨key⟩ ⟨value⟩`

`fstream` Two brace groups: ⟨*file name*⟩ and ⟨*file content*⟩

`stream` Two brace groups: ⟨*attributes (dictionary)*⟩ and ⟨*stream contents*⟩

`\pdf_object_ref:n` ⋆
New: 2021-02-10

`\pdf_object_ref:n {⟨object⟩}`

Inserts the appropriate information to reference the ⟨*object*⟩ in for example page resource allocation. If the ⟨*object*⟩ does not exist then the function expands to a reference to object zero; no PDF indirect object ever has this number, so this is a marker for error.

`\pdf_object_if_exist_p:n` ⋆
`\pdf_object_if_exist:nTF` ⋆
New: 2020-05-15

`\pdf_object_if_exist_p:n {⟨object⟩}`
`\pdf_object_if_exist:nTF {⟨object⟩} {⟨true code⟩} {⟨false code⟩}`

Tests whether an object with name {⟨*object*⟩} has been defined.

38.1.2 Indexed objects

Objects can also be created using a pair of ⟨*class*⟩ and *index*; the ⟨*class*⟩ argument should expand to character tokens, whilst the ⟨*index*⟩ is an ⟨*int expr*⟩ and starts at 1. For large families of objects, this approach is more efficient than using individual names.

`\pdf_object_new_indexed:nn`

New: 2024-04-01

`\pdf_object_new_indexed:nn {⟨class⟩} {⟨index⟩}`

Declares a PDF object of ⟨*class*⟩ and ⟨*index*⟩. The object may be referenced from this point on, and written later using `\pdf_object_write_indexed:nnnn`.

`\pdf_object_write_indexed:nnnn`
`\pdf_object_write_indexed:nnne`

New: 2024-04-01

`\pdf_object_write_indexed:nnnn {⟨class⟩} {⟨index⟩} {⟨type⟩} {⟨content⟩}`

Writes the ⟨*content*⟩ as content of the object of ⟨*class*⟩ and ⟨*index*⟩. Depending on the ⟨*type*⟩ declared for the object, the format required for the ⟨*content*⟩ will vary

`array` A space-separated list of values

`dict` Key–value pairs in the form /⟨*key*⟩ ⟨*value*⟩

`fstream` Two brace groups: ⟨*file name*⟩ and ⟨*file content*⟩

`stream` Two brace groups: ⟨*attributes (dictionary)*⟩ and ⟨*stream contents*⟩

`\pdf_object_ref_indexed:nn` ⋆

New: 2024-04-01

`\pdf_object_ref_indexed:nn {⟨class⟩} {⟨index⟩}`

Inserts the appropriate information to reference the object of ⟨*class*⟩ and ⟨*index*⟩ in for example page resource allocation. If the ⟨*class*⟩/⟨*index*⟩ combination does not exist then the function expands to a reference to object zero; no PDF indirect object ever has this number, so this is a marker for error.

38.1.3 General functions

`\pdf_object_unnamed_write:nn`
`\pdf_object_unnamed_write:ne`

New: 2021-02-10

`\pdf_object_unnamed_write:nn {⟨type⟩} {⟨content⟩}`

Writes the ⟨*content*⟩ as content of an anonymous object. Depending on the ⟨*type*⟩, the format required for ⟨*content*⟩ will vary:

`array` A space-separated list of values

`dict` Key–value pairs in the form /⟨*key*⟩ ⟨*value*⟩

`fstream` Two brace groups: ⟨*attributes (dictionary)*⟩ and ⟨*file name*⟩

`stream` Two brace groups: ⟨*attributes (dictionary)*⟩ and ⟨*stream contents*⟩

\pdf_object_ref_last: ⋆ \pdf_object_ref_last:

New: 2021-02-10 Inserts the appropriate information to reference the last ⟨*object*⟩ created. This is particularly useful for anonymous objects.

\pdf_pageobject_ref:n ⋆ \pdf_pageobject_ref:n {⟨*abspage*⟩}

New: 2021-02-10
Updated: 2024-04-22 Inserts the appropriate information to reference the ⟨*abspage*⟩; the latter is expanded fully before further processing.

38.2 Version

\pdf_version_compare_p:Nn ⋆
\pdf_version_compare:NnTF ⋆

\pdf_version_compare_p:Nn ⟨*relation*⟩ {⟨*version*⟩}
\pdf_version_compare:NnTF ⟨*relation*⟩ {⟨*version*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

New: 2021-02-10

Compares the version of the PDF being created with the ⟨*version*⟩ string specified, using the ⟨*relation*⟩. Either the ⟨*true code*⟩ or ⟨*false code*⟩ will be left in the output stream.

\pdf_version_gset:n
\pdf_version_min_gset:n

New: 2021-02-10

\pdf_version_gset:n {⟨*version*⟩}

Sets the ⟨*version*⟩ of the PDF being created. The min version will not alter the output version unless it is currently lower than the ⟨*version*⟩ requested.

This function may only be used up to the point where the PDF file is initialised. With dvips it sets \pdf_version_major: and \pdf_version_minor: and allows to compare the values with \pdf_version_compare:Nn, but the PDF version itself still has to be set with the command line option -dCompatibilityLevel of ps2pdf.

\pdf_version: ⋆
\pdf_version_major: ⋆
\pdf_version_minor: ⋆

New: 2021-02-10

\pdf_version:

Expands to the currently-active PDF version.

38.3 Page (media) size

\pdf_pagesize_gset:nn \pdf_pagesize_gset:nn {⟨*width*⟩} {⟨*height*⟩}

New: 2023-01-14 Sets the page size (mediabox) of the PDF being created to the ⟨*width*⟩ and ⟨*height*⟩, both of which are ⟨*dimexpr*⟩. The page size can only be set at the start of the output with dvips; with other backends, this can be adjusted on a per-page basis.

38.4 Compression

\pdf_uncompress: \pdf_uncompress:

New: 2021-02-10 Disables any compression of the PDF, where possible.

This function may only be used up to the point where the PDF file is initialised.

38.5 Destinations

Destinations are the places a link jumped to. Unlike the name may suggest, they don't describe an exact location in the PDF. Instead, a destination contains a reference to a page along with an instruction how to display this page. The normally used "XYZ *top left zoom*" for example instructs the viewer to show the page with the given *zoom* and the top left corner at the *top left* coordinates—which then gives the impression that there is an anchor at this position.

If an instruction takes a coordinate, it is calculated by the following commands relative to the location the command is issued. So to get a specific coordinate one has to move the command to the right place.

`\pdf_destination:nn`

New: 2021-01-03

`\pdf_destination:nn {⟨name⟩} {⟨type or integer⟩}`

This creates a destination. `{⟨type or integer⟩}` can be one of `fit`, `fith`, `fitv`, `fitb`, `fitbh`, `fitbv`, `fitr`, `xyz` or an integer representing a scale factor in percent. `fitr` here gives only a lightweight version of `/FitR`: The backend code defines `fitr` so that it will with pdfLaTeX and LuaLaTeX use the coordinates of the surrounding box, with `dvips` and `dvipdfmx` it falls back to `fit`. For full control use `\pdf_destination:nnnn`.

The keywords match to the PDF names as described in the following tabular.

Keyword	PDF	Remarks
`fit`	`/Fit`	Fits the page to the window
`fith`	`/FitH` *top*	Fits the width of the page to the window
`fitv`	`/FitV` *left*	Fits the height of the page to the window
`fitb`	`/FitB`	Fits the page bounding box to the window
`fitbh`	`/FitBH` *top*	Fits the width of the page bounding box to the window.
`fitbv`	`/FitBV` *left*	Fits the height of the page bounding box to the window.
`fitr`	`/FitR` *left bottom right top*	Fits the rectangle specified by the four coordinates to the window (see above for the restrictions)
`xyz`	`/XYZ` *left top* null	Sets a coordinate but doesn't change the zoom.
`{⟨integer⟩}`	`/XYZ` *left top zoom*	Sets a coordinate and a zoom meaning `{⟨integer⟩}`%.

`\pdf_destination:nnnn`

New: 2021-01-17

`\pdf_destination:nnnn {⟨name⟩} {⟨width⟩} {⟨height⟩} {⟨depth⟩}`

This creates a destination with `/FitR` type with the given dimensions relative to the current location. The destination is in a box of size zero, but it doesn't switch to horizontal mode.

Part VII
Implementation

Chapter 39

l3bootstrap implementation

```
⟨*package⟩
⟨@@=kernel⟩
```

39.1 The \pdfstrcmp primitive in XeTeX

Only pdfTeX has a primitive called `\pdfstrcmp`. The XeTeX version is just `\strcmp`, so there is some shuffling to do. As this is still a real primitive, using the pdfTeX name is "safe".

```
\begingroup\expandafter\expandafter\expandafter\endgroup
  \expandafter\ifx\csname pdfstrcmp\endcsname\relax
  \let\pdfstrcmp\strcmp
\fi
```

39.2 Loading support Lua code

When LuaTeX is used there are various pieces of Lua code which need to be loaded. The code itself is defined in l3luatex and is extracted into a separate file. Thus here the task is to load the Lua code both now and (if required) at the start of each job.

```
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname directlua\endcsname\relax
\else
  \ifnum\luatexversion<110 %
  \else
```

For LuaTeX we make sure the basic support is loaded: this is only necessary in plain.

```
    \begingroup\expandafter\expandafter\expandafter\endgroup
    \expandafter\ifx\csname newcatcodetable\endcsname\relax
      \input{ltluatex}%
    \fi
    \begingroup\expandafter\expandafter\expandafter\endgroup
    \expandafter\ifx\csname newluabytecode\endcsname\relax
    \else
      \newluabytecode\@expl@luadata@bytecode
    \fi
    \directlua{require("expl3")}%
```

As the user might be making a custom format, no assumption is made about matching package mode with only loading the Lua code once. Instead, a query to Lua reveals what mode is in operation.

```
    \ifnum 0%
      \directlua{
        if status.ini_version then
          tex.write("1")
        end
      }>0 %
      \everyjob\expandafter{%
        \the\expandafter\everyjob
        \csname\detokenize{lua_now:n}\endcsname{require("expl3")}%
      }%
    \fi
  \fi
\fi
```

39.3 Engine requirements

The code currently requires ε-TeX, the set of “pdfTeX extensions” *including* `\expanded`, and for Unicode engines the ability to generate arbitrary character tokens by expansion. That is covered by all supported engines since TeX Live 2019, which we therefore use as a baseline for engine and LaTeX format support. For LuaTeX, we require at least Lua 5.3 and the `token.set_lua` function. This is available at least since LuaTeX 1.10, which again is the one in TeX Live 2019. (u)pTeX only gained `\ifincsname` for TeX Live 2020, but at present that primitive is unused in expl3 so for the present it’s not tested. If and when that changes, we will need to revisit the code here.

```
\begingroup
  \def\next{\endgroup}%
  \def\ShortText{Required primitives not found}%
  \def\LongText%
    {%
      The L3 programming layer requires the e-TeX primitives and the
      \LineBreak 'pdfTeX utilities' as described in the README file.
      \LineBreak
      These are available in the engines\LineBreak
      - pdfTeX v1.40.20\LineBreak
      - XeTeX v0.999991\LineBreak
      - LuaTeX v1.10\LineBreak
      - e-(u)pTeX v3.8.2\LineBreak
      - Prote (2021)\LineBreak
      or later.\LineBreak
      \LineBreak
    }%
  \ifnum0%
    \expandafter\ifx\csname luatexversion\endcsname\relax
      \expandafter\ifx\csname expanded\endcsname\relax\else 1\fi
    \else
      \ifnum\luatexversion<110 \else 1\fi
    \fi
    =0 %
      \newlinechar`\^^J %
```

```
      \def\LineBreak{\noexpand\MessageBreak}%
      \expandafter\ifx\csname PackageError\endcsname\relax
        \def\LineBreak{^^J}%
        \begingroup
          \lccode`\~=`\ \lccode`\}=`\ %
          \lccode`\T=`\T\lccode`\H=`\H%
          \catcode`\ =11 %
\lowercase{\endgroup\def\PackageError#1#2#3{%
\begingroup\errorcontextlines-1\immediate\write0{}\errhelp{#3}\def%
\                                                    {#1 Error: #2.^^J^^J
Type  H <return>  for immediate help}\def~{\errmessage{%
\                                                    }}~\endgroup}}%
      \fi
      \edef\next
        {%
          \noexpand\PackageError{expl3}{\ShortText}
            {\LongText Loading of expl3 will abort!}%
          \endgroup
          \noexpand\endinput
        }%
  \fi
\next
```

39.4 The LaTeX3 code environment

The code environment is now set up.

\ExplSyntaxOff

Before changing any category codes, in package mode we need to save the situation before loading. Note the set up here means that once applied `\ExplSyntaxOff` becomes a “do nothing” command until `\ExplSyntaxOn` is used.

```
\protected\edef\ExplSyntaxOff
  {%
    \protected\def\noexpand\ExplSyntaxOff{}%
    \catcode   9 = \the\catcode   9\relax
    \catcode  32 = \the\catcode  32\relax
    \catcode  34 = \the\catcode  34\relax
    \catcode  58 = \the\catcode  58\relax
    \catcode  94 = \the\catcode  94\relax
    \catcode  95 = \the\catcode  95\relax
    \catcode 124 = \the\catcode 124\relax
    \catcode 126 = \the\catcode 126\relax
    \endlinechar = \the\endlinechar\relax
    \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0\relax
  }%
```

(*End of definition for* `\ExplSyntaxOff`. *This function is documented on page 10.*)

The code environment is now set up.

```
\catcode 9   = 9\relax
\catcode 32  = 9\relax
\catcode 34  = 12\relax
\catcode 58  = 11\relax
\catcode 94  = 7\relax
\catcode 95  = 11\relax
```

```
\catcode 124 = 12\relax
\catcode 126 = 10\relax
\endlinechar = 32\relax
```

\l__kernel_expl_bool The status for code syntax: this is on at present.

```
\global\chardef\l__kernel_expl_bool = 1\relax
```

(*End of definition for* `\l__kernel_expl_bool`.)

\ExplSyntaxOn The idea here is that multiple `\ExplSyntaxOn` calls are not going to mess up category codes, and that multiple calls to `\ExplSyntaxOff` are also not wasting time. Applying `\ExplSyntaxOn` alters the definition of `\ExplSyntaxOff` and so in package mode this function should not be used until after the end of the loading process!

```
\protected \def \ExplSyntaxOn
  {
    \bool_if:NF \l__kernel_expl_bool
      {
        \cs_set_protected:Npe \ExplSyntaxOff
          {
            \char_set_catcode:nn { 9 }   { \char_value_catcode:n { 9 } }
            \char_set_catcode:nn { 32 }  { \char_value_catcode:n { 32 } }
            \char_set_catcode:nn { 34 }  { \char_value_catcode:n { 34 } }
            \char_set_catcode:nn { 58 }  { \char_value_catcode:n { 58 } }
            \char_set_catcode:nn { 94 }  { \char_value_catcode:n { 94 } }
            \char_set_catcode:nn { 95 }  { \char_value_catcode:n { 95 } }
            \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
            \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
            \tex_endlinechar:D =
              \tex_the:D \tex_endlinechar:D \scan_stop:
            \bool_set_false:N \l__kernel_expl_bool
            \cs_set_protected:Npn \ExplSyntaxOff { }
          }
      }
    \char_set_catcode_ignore:n           { 9 }   % tab
    \char_set_catcode_ignore:n           { 32 }  % space
    \char_set_catcode_other:n            { 34 }  % double quote
    \char_set_catcode_letter:n           { 58 }  % colon
    \char_set_catcode_math_superscript:n { 94 }  % circumflex
    \char_set_catcode_letter:n           { 95 }  % underscore
    \char_set_catcode_other:n            { 124 } % pipe
    \char_set_catcode_space:n            { 126 } % tilde
    \tex_endlinechar:D = 32 \scan_stop:
    \bool_set_true:N \l__kernel_expl_bool
  }
```

(*End of definition for* `\ExplSyntaxOn`. *This function is documented on page 10.*)

⟨/package⟩

Chapter 40

l3names implementation

```
⟨*package & tex⟩
```

The prefix here is `kernel`. A few places need `@@` to be left as is; this is obtained as `@@@@`.

```
⟨@@=kernel⟩
```

The code here simply renames all of the primitives to new, internal, names.

The `\let` primitive is renamed by hand first as it is essential for the entire process to follow. This also uses `\global`, as that way we avoid leaving an unneeded csname in the hash table.

```
\let \tex_global:D \global
\let \tex_let:D    \let
```

Everything is inside a (rather long) group, which keeps `\__kernel_primitive:NN` trapped.

```
\begingroup
```

`\__kernel_primitive:NN` A temporary function to actually do the renaming.

```
  \long \def \__kernel_primitive:NN #1#2
    { \tex_global:D \tex_let:D #2 #1 }
```

(*End of definition for* `\__kernel_primitive:NN`.)

To allow extracting "just the names", a bit of DocStrip fiddling.

```
⟨/package & tex⟩
⟨*names | tex⟩
⟨*names | package⟩
```

In the current incarnation of this module, all TeX primitives are given a new name of the form `\tex_`*oldname*`:D`. But first three special cases which have symbolic original names. These are given modified new names, so that they may be entered without catcode tricks.

```
  \__kernel_primitive:NN \                        \tex_space:D
  \__kernel_primitive:NN \/                       \tex_italiccorrection:D
  \__kernel_primitive:NN \-                       \tex_hyphen:D
```

Now all the other primitives.

```
  \__kernel_primitive:NN \above                   \tex_above:D
  \__kernel_primitive:NN \abovedisplayshortskip   \tex_abovedisplayshortskip:D
  \__kernel_primitive:NN \abovedisplayskip        \tex_abovedisplayskip:D
  \__kernel_primitive:NN \abovewithdelims         \tex_abovewithdelims:D
```

```
\__kernel_primitive:NN \accent                  \tex_accent:D
\__kernel_primitive:NN \adjdemerits             \tex_adjdemerits:D
\__kernel_primitive:NN \advance                 \tex_advance:D
\__kernel_primitive:NN \afterassignment         \tex_afterassignment:D
\__kernel_primitive:NN \aftergroup              \tex_aftergroup:D
\__kernel_primitive:NN \atop                    \tex_atop:D
\__kernel_primitive:NN \atopwithdelims          \tex_atopwithdelims:D
\__kernel_primitive:NN \badness                 \tex_badness:D
\__kernel_primitive:NN \baselineskip            \tex_baselineskip:D
\__kernel_primitive:NN \batchmode               \tex_batchmode:D
\__kernel_primitive:NN \begingroup              \tex_begingroup:D
\__kernel_primitive:NN \belowdisplayshortskip   \tex_belowdisplayshortskip:D
\__kernel_primitive:NN \belowdisplayskip        \tex_belowdisplayskip:D
\__kernel_primitive:NN \binoppenalty            \tex_binoppenalty:D
\__kernel_primitive:NN \botmark                 \tex_botmark:D
\__kernel_primitive:NN \box                     \tex_box:D
\__kernel_primitive:NN \boxmaxdepth             \tex_boxmaxdepth:D
\__kernel_primitive:NN \brokenpenalty           \tex_brokenpenalty:D
\__kernel_primitive:NN \catcode                 \tex_catcode:D
\__kernel_primitive:NN \char                    \tex_char:D
\__kernel_primitive:NN \chardef                 \tex_chardef:D
\__kernel_primitive:NN \cleaders                \tex_cleaders:D
\__kernel_primitive:NN \closein                 \tex_closein:D
\__kernel_primitive:NN \closeout                \tex_closeout:D
\__kernel_primitive:NN \clubpenalty             \tex_clubpenalty:D
\__kernel_primitive:NN \copy                    \tex_copy:D
\__kernel_primitive:NN \count                   \tex_count:D
\__kernel_primitive:NN \countdef                \tex_countdef:D
\__kernel_primitive:NN \cr                      \tex_cr:D
\__kernel_primitive:NN \crcr                    \tex_crcr:D
\__kernel_primitive:NN \csname                  \tex_csname:D
\__kernel_primitive:NN \day                     \tex_day:D
\__kernel_primitive:NN \deadcycles              \tex_deadcycles:D
\__kernel_primitive:NN \def                     \tex_def:D
\__kernel_primitive:NN \defaulthyphenchar       \tex_defaulthyphenchar:D
\__kernel_primitive:NN \defaultskewchar         \tex_defaultskewchar:D
\__kernel_primitive:NN \delcode                 \tex_delcode:D
\__kernel_primitive:NN \delimiter               \tex_delimiter:D
\__kernel_primitive:NN \delimiterfactor         \tex_delimiterfactor:D
\__kernel_primitive:NN \delimitershortfall      \tex_delimitershortfall:D
\__kernel_primitive:NN \dimen                   \tex_dimen:D
\__kernel_primitive:NN \dimendef                \tex_dimendef:D
\__kernel_primitive:NN \discretionary           \tex_discretionary:D
\__kernel_primitive:NN \displayindent           \tex_displayindent:D
\__kernel_primitive:NN \displaylimits           \tex_displaylimits:D
\__kernel_primitive:NN \displaystyle            \tex_displaystyle:D
\__kernel_primitive:NN \displaywidowpenalty     \tex_displaywidowpenalty:D
\__kernel_primitive:NN \displaywidth            \tex_displaywidth:D
\__kernel_primitive:NN \divide                  \tex_divide:D
\__kernel_primitive:NN \doublehyphendemerits    \tex_doublehyphendemerits:D
\__kernel_primitive:NN \dp                      \tex_dp:D
\__kernel_primitive:NN \dump                    \tex_dump:D
\__kernel_primitive:NN \edef                    \tex_edef:D
\__kernel_primitive:NN \else                    \tex_else:D
```

```
\__kernel_primitive:NN \emergencystretch      \tex_emergencystretch:D
\__kernel_primitive:NN \end                   \tex_end:D
\__kernel_primitive:NN \endcsname             \tex_endcsname:D
\__kernel_primitive:NN \endgroup              \tex_endgroup:D
\__kernel_primitive:NN \endinput              \tex_endinput:D
\__kernel_primitive:NN \endlinechar           \tex_endlinechar:D
\__kernel_primitive:NN \eqno                  \tex_eqno:D
\__kernel_primitive:NN \errhelp               \tex_errhelp:D
\__kernel_primitive:NN \errmessage            \tex_errmessage:D
\__kernel_primitive:NN \errorcontextlines     \tex_errorcontextlines:D
\__kernel_primitive:NN \errorstopmode         \tex_errorstopmode:D
\__kernel_primitive:NN \escapechar            \tex_escapechar:D
\__kernel_primitive:NN \everycr               \tex_everycr:D
\__kernel_primitive:NN \everydisplay          \tex_everydisplay:D
\__kernel_primitive:NN \everyhbox             \tex_everyhbox:D
\__kernel_primitive:NN \everyjob              \tex_everyjob:D
\__kernel_primitive:NN \everymath             \tex_everymath:D
\__kernel_primitive:NN \everypar              \tex_everypar:D
\__kernel_primitive:NN \everyvbox             \tex_everyvbox:D
\__kernel_primitive:NN \exhyphenpenalty       \tex_exhyphenpenalty:D
\__kernel_primitive:NN \expandafter           \tex_expandafter:D
\__kernel_primitive:NN \fam                   \tex_fam:D
\__kernel_primitive:NN \fi                    \tex_fi:D
\__kernel_primitive:NN \finalhyphendemerits   \tex_finalhyphendemerits:D
\__kernel_primitive:NN \firstmark             \tex_firstmark:D
\__kernel_primitive:NN \floatingpenalty       \tex_floatingpenalty:D
\__kernel_primitive:NN \font                  \tex_font:D
\__kernel_primitive:NN \fontdimen             \tex_fontdimen:D
\__kernel_primitive:NN \fontname              \tex_fontname:D
\__kernel_primitive:NN \futurelet             \tex_futurelet:D
\__kernel_primitive:NN \gdef                  \tex_gdef:D
\__kernel_primitive:NN \global                \tex_global:D
\__kernel_primitive:NN \globaldefs            \tex_globaldefs:D
\__kernel_primitive:NN \halign                \tex_halign:D
\__kernel_primitive:NN \hangafter             \tex_hangafter:D
\__kernel_primitive:NN \hangindent            \tex_hangindent:D
\__kernel_primitive:NN \hbadness              \tex_hbadness:D
\__kernel_primitive:NN \hbox                  \tex_hbox:D
\__kernel_primitive:NN \hfil                  \tex_hfil:D
\__kernel_primitive:NN \hfill                 \tex_hfill:D
\__kernel_primitive:NN \hfilneg               \tex_hfilneg:D
\__kernel_primitive:NN \hfuzz                 \tex_hfuzz:D
\__kernel_primitive:NN \hoffset               \tex_hoffset:D
\__kernel_primitive:NN \holdinginserts        \tex_holdinginserts:D
\__kernel_primitive:NN \hrule                 \tex_hrule:D
\__kernel_primitive:NN \hsize                 \tex_hsize:D
\__kernel_primitive:NN \hskip                 \tex_hskip:D
\__kernel_primitive:NN \hss                   \tex_hss:D
\__kernel_primitive:NN \ht                    \tex_ht:D
\__kernel_primitive:NN \hyphenation           \tex_hyphenation:D
\__kernel_primitive:NN \hyphenchar            \tex_hyphenchar:D
\__kernel_primitive:NN \hyphenpenalty         \tex_hyphenpenalty:D
\__kernel_primitive:NN \if                    \tex_if:D
\__kernel_primitive:NN \ifcase                \tex_ifcase:D
```

```
\__kernel_primitive:NN \ifcat              \tex_ifcat:D
\__kernel_primitive:NN \ifdim              \tex_ifdim:D
\__kernel_primitive:NN \ifeof              \tex_ifeof:D
\__kernel_primitive:NN \iffalse            \tex_iffalse:D
\__kernel_primitive:NN \ifhbox             \tex_ifhbox:D
\__kernel_primitive:NN \ifhmode            \tex_ifhmode:D
\__kernel_primitive:NN \ifinner            \tex_ifinner:D
\__kernel_primitive:NN \ifmmode            \tex_ifmmode:D
\__kernel_primitive:NN \ifnum              \tex_ifnum:D
\__kernel_primitive:NN \ifodd              \tex_ifodd:D
\__kernel_primitive:NN \iftrue             \tex_iftrue:D
\__kernel_primitive:NN \ifvbox             \tex_ifvbox:D
\__kernel_primitive:NN \ifvmode            \tex_ifvmode:D
\__kernel_primitive:NN \ifvoid             \tex_ifvoid:D
\__kernel_primitive:NN \ifx                \tex_ifx:D
\__kernel_primitive:NN \ignorespaces       \tex_ignorespaces:D
\__kernel_primitive:NN \immediate          \tex_immediate:D
\__kernel_primitive:NN \indent             \tex_indent:D
\__kernel_primitive:NN \input              \tex_input:D
\__kernel_primitive:NN \inputlineno        \tex_inputlineno:D
\__kernel_primitive:NN \insert             \tex_insert:D
\__kernel_primitive:NN \insertpenalties    \tex_insertpenalties:D
\__kernel_primitive:NN \interlinepenalty   \tex_interlinepenalty:D
\__kernel_primitive:NN \jobname            \tex_jobname:D
\__kernel_primitive:NN \kern               \tex_kern:D
\__kernel_primitive:NN \language           \tex_language:D
\__kernel_primitive:NN \lastbox            \tex_lastbox:D
\__kernel_primitive:NN \lastkern           \tex_lastkern:D
\__kernel_primitive:NN \lastpenalty        \tex_lastpenalty:D
\__kernel_primitive:NN \lastskip           \tex_lastskip:D
\__kernel_primitive:NN \lccode             \tex_lccode:D
\__kernel_primitive:NN \leaders            \tex_leaders:D
\__kernel_primitive:NN \left               \tex_left:D
\__kernel_primitive:NN \lefthyphenmin      \tex_lefthyphenmin:D
\__kernel_primitive:NN \leftskip           \tex_leftskip:D
\__kernel_primitive:NN \leqno              \tex_leqno:D
\__kernel_primitive:NN \let                \tex_let:D
\__kernel_primitive:NN \limits             \tex_limits:D
\__kernel_primitive:NN \linepenalty        \tex_linepenalty:D
\__kernel_primitive:NN \lineskip           \tex_lineskip:D
\__kernel_primitive:NN \lineskiplimit      \tex_lineskiplimit:D
\__kernel_primitive:NN \long               \tex_long:D
\__kernel_primitive:NN \looseness          \tex_looseness:D
\__kernel_primitive:NN \lower              \tex_lower:D
\__kernel_primitive:NN \lowercase          \tex_lowercase:D
\__kernel_primitive:NN \mag                \tex_mag:D
\__kernel_primitive:NN \mark               \tex_mark:D
\__kernel_primitive:NN \mathaccent         \tex_mathaccent:D
\__kernel_primitive:NN \mathbin            \tex_mathbin:D
\__kernel_primitive:NN \mathchar           \tex_mathchar:D
\__kernel_primitive:NN \mathchardef        \tex_mathchardef:D
\__kernel_primitive:NN \mathchoice         \tex_mathchoice:D
\__kernel_primitive:NN \mathclose          \tex_mathclose:D
\__kernel_primitive:NN \mathcode           \tex_mathcode:D
```

```
\__kernel_primitive:NN \mathinner           \tex_mathinner:D
\__kernel_primitive:NN \mathop              \tex_mathop:D
\__kernel_primitive:NN \mathopen            \tex_mathopen:D
\__kernel_primitive:NN \mathord             \tex_mathord:D
\__kernel_primitive:NN \mathpunct           \tex_mathpunct:D
\__kernel_primitive:NN \mathrel             \tex_mathrel:D
\__kernel_primitive:NN \mathsurround        \tex_mathsurround:D
\__kernel_primitive:NN \maxdeadcycles       \tex_maxdeadcycles:D
\__kernel_primitive:NN \maxdepth            \tex_maxdepth:D
\__kernel_primitive:NN \meaning             \tex_meaning:D
\__kernel_primitive:NN \medmuskip           \tex_medmuskip:D
\__kernel_primitive:NN \message             \tex_message:D
\__kernel_primitive:NN \mkern               \tex_mkern:D
\__kernel_primitive:NN \month               \tex_month:D
\__kernel_primitive:NN \moveleft            \tex_moveleft:D
\__kernel_primitive:NN \moveright           \tex_moveright:D
\__kernel_primitive:NN \mskip               \tex_mskip:D
\__kernel_primitive:NN \multiply            \tex_multiply:D
\__kernel_primitive:NN \muskip              \tex_muskip:D
\__kernel_primitive:NN \muskipdef           \tex_muskipdef:D
\__kernel_primitive:NN \newlinechar         \tex_newlinechar:D
\__kernel_primitive:NN \noalign             \tex_noalign:D
\__kernel_primitive:NN \noboundary          \tex_noboundary:D
\__kernel_primitive:NN \noexpand            \tex_noexpand:D
\__kernel_primitive:NN \noindent            \tex_noindent:D
\__kernel_primitive:NN \nolimits            \tex_nolimits:D
\__kernel_primitive:NN \nonscript           \tex_nonscript:D
\__kernel_primitive:NN \nonstopmode         \tex_nonstopmode:D
\__kernel_primitive:NN \nulldelimiterspace  \tex_nulldelimiterspace:D
\__kernel_primitive:NN \nullfont            \tex_nullfont:D
\__kernel_primitive:NN \number              \tex_number:D
\__kernel_primitive:NN \omit                \tex_omit:D
\__kernel_primitive:NN \openin              \tex_openin:D
\__kernel_primitive:NN \openout             \tex_openout:D
\__kernel_primitive:NN \or                  \tex_or:D
\__kernel_primitive:NN \outer               \tex_outer:D
\__kernel_primitive:NN \output              \tex_output:D
\__kernel_primitive:NN \outputpenalty       \tex_outputpenalty:D
\__kernel_primitive:NN \over                \tex_over:D
\__kernel_primitive:NN \overfullrule        \tex_overfullrule:D
\__kernel_primitive:NN \overline            \tex_overline:D
\__kernel_primitive:NN \overwithdelims      \tex_overwithdelims:D
\__kernel_primitive:NN \pagedepth           \tex_pagedepth:D
\__kernel_primitive:NN \pagefilllstretch    \tex_pagefilllstretch:D
\__kernel_primitive:NN \pagefillstretch     \tex_pagefillstretch:D
\__kernel_primitive:NN \pagefilstretch      \tex_pagefilstretch:D
\__kernel_primitive:NN \pagegoal            \tex_pagegoal:D
\__kernel_primitive:NN \pageshrink          \tex_pageshrink:D
\__kernel_primitive:NN \pagestretch         \tex_pagestretch:D
\__kernel_primitive:NN \pagetotal           \tex_pagetotal:D
\__kernel_primitive:NN \par                 \tex_par:D
\__kernel_primitive:NN \parfillskip         \tex_parfillskip:D
\__kernel_primitive:NN \parindent           \tex_parindent:D
\__kernel_primitive:NN \parshape            \tex_parshape:D
```

```
\__kernel_primitive:NN \parskip                \tex_parskip:D
\__kernel_primitive:NN \patterns               \tex_patterns:D
\__kernel_primitive:NN \pausing                \tex_pausing:D
\__kernel_primitive:NN \penalty                \tex_penalty:D
\__kernel_primitive:NN \postdisplaypenalty     \tex_postdisplaypenalty:D
\__kernel_primitive:NN \predisplaypenalty      \tex_predisplaypenalty:D
\__kernel_primitive:NN \predisplaysize         \tex_predisplaysize:D
\__kernel_primitive:NN \pretolerance           \tex_pretolerance:D
\__kernel_primitive:NN \prevdepth              \tex_prevdepth:D
\__kernel_primitive:NN \prevgraf               \tex_prevgraf:D
\__kernel_primitive:NN \radical                \tex_radical:D
\__kernel_primitive:NN \raise                  \tex_raise:D
\__kernel_primitive:NN \read                   \tex_read:D
\__kernel_primitive:NN \relax                  \tex_relax:D
\__kernel_primitive:NN \relpenalty             \tex_relpenalty:D
\__kernel_primitive:NN \right                  \tex_right:D
\__kernel_primitive:NN \righthyphenmin         \tex_righthyphenmin:D
\__kernel_primitive:NN \rightskip              \tex_rightskip:D
\__kernel_primitive:NN \romannumeral           \tex_romannumeral:D
\__kernel_primitive:NN \scriptfont             \tex_scriptfont:D
\__kernel_primitive:NN \scriptscriptfont       \tex_scriptscriptfont:D
\__kernel_primitive:NN \scriptscriptstyle      \tex_scriptscriptstyle:D
\__kernel_primitive:NN \scriptspace            \tex_scriptspace:D
\__kernel_primitive:NN \scriptstyle            \tex_scriptstyle:D
\__kernel_primitive:NN \scrollmode             \tex_scrollmode:D
\__kernel_primitive:NN \setbox                 \tex_setbox:D
\__kernel_primitive:NN \setlanguage            \tex_setlanguage:D
\__kernel_primitive:NN \sfcode                 \tex_sfcode:D
\__kernel_primitive:NN \shipout                \tex_shipout:D
\__kernel_primitive:NN \show                   \tex_show:D
\__kernel_primitive:NN \showbox                \tex_showbox:D
\__kernel_primitive:NN \showboxbreadth         \tex_showboxbreadth:D
\__kernel_primitive:NN \showboxdepth           \tex_showboxdepth:D
\__kernel_primitive:NN \showlists              \tex_showlists:D
\__kernel_primitive:NN \showthe                \tex_showthe:D
\__kernel_primitive:NN \skewchar               \tex_skewchar:D
\__kernel_primitive:NN \skip                   \tex_skip:D
\__kernel_primitive:NN \skipdef                \tex_skipdef:D
\__kernel_primitive:NN \spacefactor            \tex_spacefactor:D
\__kernel_primitive:NN \spaceskip              \tex_spaceskip:D
\__kernel_primitive:NN \span                   \tex_span:D
\__kernel_primitive:NN \special                \tex_special:D
\__kernel_primitive:NN \splitbotmark           \tex_splitbotmark:D
\__kernel_primitive:NN \splitfirstmark         \tex_splitfirstmark:D
\__kernel_primitive:NN \splitmaxdepth          \tex_splitmaxdepth:D
\__kernel_primitive:NN \splittopskip           \tex_splittopskip:D
\__kernel_primitive:NN \string                 \tex_string:D
\__kernel_primitive:NN \tabskip                \tex_tabskip:D
\__kernel_primitive:NN \textfont               \tex_textfont:D
\__kernel_primitive:NN \textstyle              \tex_textstyle:D
\__kernel_primitive:NN \the                    \tex_the:D
\__kernel_primitive:NN \thickmuskip            \tex_thickmuskip:D
\__kernel_primitive:NN \thinmuskip             \tex_thinmuskip:D
\__kernel_primitive:NN \time                   \tex_time:D
```

```
\__kernel_primitive:NN \toks                \tex_toks:D
\__kernel_primitive:NN \toksdef             \tex_toksdef:D
\__kernel_primitive:NN \tolerance           \tex_tolerance:D
\__kernel_primitive:NN \topmark             \tex_topmark:D
\__kernel_primitive:NN \topskip             \tex_topskip:D
\__kernel_primitive:NN \tracingcommands     \tex_tracingcommands:D
\__kernel_primitive:NN \tracinglostchars    \tex_tracinglostchars:D
\__kernel_primitive:NN \tracingmacros       \tex_tracingmacros:D
\__kernel_primitive:NN \tracingonline       \tex_tracingonline:D
\__kernel_primitive:NN \tracingoutput       \tex_tracingoutput:D
\__kernel_primitive:NN \tracingpages        \tex_tracingpages:D
\__kernel_primitive:NN \tracingparagraphs   \tex_tracingparagraphs:D
\__kernel_primitive:NN \tracingrestores     \tex_tracingrestores:D
\__kernel_primitive:NN \tracingstats        \tex_tracingstats:D
\__kernel_primitive:NN \uccode              \tex_uccode:D
\__kernel_primitive:NN \uchyph              \tex_uchyph:D
\__kernel_primitive:NN \underline           \tex_underline:D
\__kernel_primitive:NN \unhbox              \tex_unhbox:D
\__kernel_primitive:NN \unhcopy             \tex_unhcopy:D
\__kernel_primitive:NN \unkern              \tex_unkern:D
\__kernel_primitive:NN \unpenalty           \tex_unpenalty:D
\__kernel_primitive:NN \unskip              \tex_unskip:D
\__kernel_primitive:NN \unvbox              \tex_unvbox:D
\__kernel_primitive:NN \unvcopy             \tex_unvcopy:D
\__kernel_primitive:NN \uppercase           \tex_uppercase:D
\__kernel_primitive:NN \vadjust             \tex_vadjust:D
\__kernel_primitive:NN \valign              \tex_valign:D
\__kernel_primitive:NN \vbadness            \tex_vbadness:D
\__kernel_primitive:NN \vbox                \tex_vbox:D
\__kernel_primitive:NN \vcenter             \tex_vcenter:D
\__kernel_primitive:NN \vfil                \tex_vfil:D
\__kernel_primitive:NN \vfill               \tex_vfill:D
\__kernel_primitive:NN \vfilneg             \tex_vfilneg:D
\__kernel_primitive:NN \vfuzz               \tex_vfuzz:D
\__kernel_primitive:NN \voffset             \tex_voffset:D
\__kernel_primitive:NN \vrule               \tex_vrule:D
\__kernel_primitive:NN \vsize               \tex_vsize:D
\__kernel_primitive:NN \vskip               \tex_vskip:D
\__kernel_primitive:NN \vsplit              \tex_vsplit:D
\__kernel_primitive:NN \vss                 \tex_vss:D
\__kernel_primitive:NN \vtop                \tex_vtop:D
\__kernel_primitive:NN \wd                  \tex_wd:D
\__kernel_primitive:NN \widowpenalty        \tex_widowpenalty:D
\__kernel_primitive:NN \write               \tex_write:D
\__kernel_primitive:NN \xdef                \tex_xdef:D
\__kernel_primitive:NN \xleaders            \tex_xleaders:D
\__kernel_primitive:NN \xspaceskip          \tex_xspaceskip:D
\__kernel_primitive:NN \year                \tex_year:D
```

Primitives introduced by ε-TEX.

```
\__kernel_primitive:NN \beginL              \tex_beginL:D
\__kernel_primitive:NN \beginR              \tex_beginR:D
\__kernel_primitive:NN \botmarks            \tex_botmarks:D
\__kernel_primitive:NN \clubpenalties       \tex_clubpenalties:D
\__kernel_primitive:NN \currentgrouplevel   \tex_currentgrouplevel:D
```

```
\__kernel_primitive:NN \currentgrouptype      \tex_currentgrouptype:D
\__kernel_primitive:NN \currentifbranch       \tex_currentifbranch:D
\__kernel_primitive:NN \currentiflevel        \tex_currentiflevel:D
\__kernel_primitive:NN \currentiftype         \tex_currentiftype:D
\__kernel_primitive:NN \detokenize            \tex_detokenize:D
\__kernel_primitive:NN \dimexpr               \tex_dimexpr:D
\__kernel_primitive:NN \displaywidowpenalties \tex_displaywidowpenalties:D
\__kernel_primitive:NN \endL                  \tex_endL:D
\__kernel_primitive:NN \endR                  \tex_endR:D
\__kernel_primitive:NN \eTeXrevision          \tex_eTeXrevision:D
\__kernel_primitive:NN \eTeXversion           \tex_eTeXversion:D
\__kernel_primitive:NN \everyeof              \tex_everyeof:D
\__kernel_primitive:NN \firstmarks            \tex_firstmarks:D
\__kernel_primitive:NN \fontchardp            \tex_fontchardp:D
\__kernel_primitive:NN \fontcharht            \tex_fontcharht:D
\__kernel_primitive:NN \fontcharic            \tex_fontcharic:D
\__kernel_primitive:NN \fontcharwd            \tex_fontcharwd:D
\__kernel_primitive:NN \glueexpr              \tex_glueexpr:D
\__kernel_primitive:NN \glueshrink            \tex_glueshrink:D
\__kernel_primitive:NN \glueshrinkorder       \tex_glueshrinkorder:D
\__kernel_primitive:NN \gluestretch           \tex_gluestretch:D
\__kernel_primitive:NN \gluestretchorder      \tex_gluestretchorder:D
\__kernel_primitive:NN \gluetomu              \tex_gluetomu:D
\__kernel_primitive:NN \ifcsname              \tex_ifcsname:D
\__kernel_primitive:NN \ifdefined             \tex_ifdefined:D
\__kernel_primitive:NN \iffontchar            \tex_iffontchar:D
\__kernel_primitive:NN \interactionmode       \tex_interactionmode:D
\__kernel_primitive:NN \interlinepenalties    \tex_interlinepenalties:D
\__kernel_primitive:NN \lastlinefit           \tex_lastlinefit:D
\__kernel_primitive:NN \lastnodetype          \tex_lastnodetype:D
\__kernel_primitive:NN \marks                 \tex_marks:D
\__kernel_primitive:NN \middle                \tex_middle:D
\__kernel_primitive:NN \muexpr                \tex_muexpr:D
\__kernel_primitive:NN \mutoglue              \tex_mutoglue:D
\__kernel_primitive:NN \numexpr               \tex_numexpr:D
\__kernel_primitive:NN \pagediscards          \tex_pagediscards:D
\__kernel_primitive:NN \parshapedimen         \tex_parshapedimen:D
\__kernel_primitive:NN \parshapeindent        \tex_parshapeindent:D
\__kernel_primitive:NN \parshapelength        \tex_parshapelength:D
\__kernel_primitive:NN \predisplaydirection   \tex_predisplaydirection:D
\__kernel_primitive:NN \protected             \tex_protected:D
\__kernel_primitive:NN \readline              \tex_readline:D
\__kernel_primitive:NN \savinghyphcodes       \tex_savinghyphcodes:D
\__kernel_primitive:NN \savingvdiscards       \tex_savingvdiscards:D
\__kernel_primitive:NN \scantokens            \tex_scantokens:D
\__kernel_primitive:NN \showgroups            \tex_showgroups:D
\__kernel_primitive:NN \showifs               \tex_showifs:D
\__kernel_primitive:NN \showtokens            \tex_showtokens:D
\__kernel_primitive:NN \splitbotmarks         \tex_splitbotmarks:D
\__kernel_primitive:NN \splitdiscards         \tex_splitdiscards:D
\__kernel_primitive:NN \splitfirstmarks       \tex_splitfirstmarks:D
\__kernel_primitive:NN \TeXXeTstate           \tex_TeXXeTstate:D
\__kernel_primitive:NN \topmarks              \tex_topmarks:D
\__kernel_primitive:NN \tracingassigns        \tex_tracingassigns:D
```

```
\__kernel_primitive:NN \tracinggroups         \tex_tracinggroups:D
\__kernel_primitive:NN \tracingifs            \tex_tracingifs:D
\__kernel_primitive:NN \tracingnesting        \tex_tracingnesting:D
\__kernel_primitive:NN \tracingscantokens     \tex_tracingscantokens:D
\__kernel_primitive:NN \unexpanded            \tex_unexpanded:D
\__kernel_primitive:NN \unless                \tex_unless:D
\__kernel_primitive:NN \widowpenalties        \tex_widowpenalties:D
```

Post-ε-TEX primitives do not always end up with the same name in all engines, if indeed they are available cross-engine anyway. We therefore take the approach of preferring the shortest name that makes sense. First, we deal with the primitives introduced by pdfTEX which directly relate to PDF output: these are copied with the names unchanged.

```
\__kernel_primitive:NN \pdfannot              \tex_pdfannot:D
\__kernel_primitive:NN \pdfcatalog            \tex_pdfcatalog:D
\__kernel_primitive:NN \pdfcompresslevel      \tex_pdfcompresslevel:D
\__kernel_primitive:NN \pdfcolorstack         \tex_pdfcolorstack:D
\__kernel_primitive:NN \pdfcolorstackinit     \tex_pdfcolorstackinit:D
\__kernel_primitive:NN \pdfdecimaldigits      \tex_pdfdecimaldigits:D
\__kernel_primitive:NN \pdfdest               \tex_pdfdest:D
\__kernel_primitive:NN \pdfdestmargin         \tex_pdfdestmargin:D
\__kernel_primitive:NN \pdfendlink            \tex_pdfendlink:D
\__kernel_primitive:NN \pdfendthread          \tex_pdfendthread:D
\__kernel_primitive:NN \pdffakespace          \tex_pdffakespace:D
\__kernel_primitive:NN \pdffontattr           \tex_pdffontattr:D
\__kernel_primitive:NN \pdffontname           \tex_pdffontname:D
\__kernel_primitive:NN \pdffontobjnum         \tex_pdffontobjnum:D
\__kernel_primitive:NN \pdfgamma              \tex_pdfgamma:D
\__kernel_primitive:NN \pdfgentounicode       \tex_pdfgentounicode:D
\__kernel_primitive:NN \pdfglyphtounicode     \tex_pdfglyphtounicode:D
\__kernel_primitive:NN \pdfhorigin            \tex_pdfhorigin:D
\__kernel_primitive:NN \pdfimageapplygamma    \tex_pdfimageapplygamma:D
\__kernel_primitive:NN \pdfimagegamma         \tex_pdfimagegamma:D
\__kernel_primitive:NN \pdfimagehicolor       \tex_pdfimagehicolor:D
\__kernel_primitive:NN \pdfimageresolution    \tex_pdfimageresolution:D
\__kernel_primitive:NN \pdfincludechars       \tex_pdfincludechars:D
\__kernel_primitive:NN \pdfinclusioncopyfonts \tex_pdfinclusioncopyfonts:D
\__kernel_primitive:NN \pdfinclusionerrorlevel
  \tex_pdfinclusionerrorlevel:D
\__kernel_primitive:NN \pdfinfo               \tex_pdfinfo:D
\__kernel_primitive:NN \pdfinfoomitdate       \tex_pdfinfoomitdate:D
\__kernel_primitive:NN \pdfinterwordspaceoff  \tex_pdfinterwordspaceoff:D
\__kernel_primitive:NN \pdfinterwordspaceon   \tex_pdfinterwordspaceon:D
\__kernel_primitive:NN \pdflastannot          \tex_pdflastannot:D
\__kernel_primitive:NN \pdflastlink           \tex_pdflastlink:D
\__kernel_primitive:NN \pdflastobj            \tex_pdflastobj:D
\__kernel_primitive:NN \pdflastxform          \tex_pdflastxform:D
\__kernel_primitive:NN \pdflastximage         \tex_pdflastximage:D
\__kernel_primitive:NN \pdflastximagecolordepth
  \tex_pdflastximagecolordepth:D
\__kernel_primitive:NN \pdflastximagepages    \tex_pdflastximagepages:D
\__kernel_primitive:NN \pdflinkmargin         \tex_pdflinkmargin:D
\__kernel_primitive:NN \pdfliteral            \tex_pdfliteral:D
\__kernel_primitive:NN \pdfmapfile            \tex_pdfmapfile:D
\__kernel_primitive:NN \pdfmapline            \tex_pdfmapline:D
```

```
\__kernel_primitive:NN \pdfmajorversion       \tex_pdfmajorversion:D
\__kernel_primitive:NN \pdfminorversion       \tex_pdfminorversion:D
\__kernel_primitive:NN \pdfnames              \tex_pdfnames:D
\__kernel_primitive:NN \pdfnobuiltintounicode \tex_pdfnobuiltintounicode:D
\__kernel_primitive:NN \pdfobj                \tex_pdfobj:D
\__kernel_primitive:NN \pdfobjcompresslevel   \tex_pdfobjcompresslevel:D
\__kernel_primitive:NN \pdfomitcharset        \tex_pdfomitcharset:D
\__kernel_primitive:NN \pdfoutline            \tex_pdfoutline:D
\__kernel_primitive:NN \pdfoutput             \tex_pdfoutput:D
\__kernel_primitive:NN \pdfpageattr           \tex_pdfpageattr:D
\__kernel_primitive:NN \pdfpagebox            \tex_pdfpagebox:D
\__kernel_primitive:NN \pdfpageref            \tex_pdfpageref:D
\__kernel_primitive:NN \pdfpageresources      \tex_pdfpageresources:D
\__kernel_primitive:NN \pdfpagesattr          \tex_pdfpagesattr:D
\__kernel_primitive:NN \pdfrefobj             \tex_pdfrefobj:D
\__kernel_primitive:NN \pdfrefxform           \tex_pdfrefxform:D
\__kernel_primitive:NN \pdfrefximage          \tex_pdfrefximage:D
\__kernel_primitive:NN \pdfrestore            \tex_pdfrestore:D
\__kernel_primitive:NN \pdfretval             \tex_pdfretval:D
\__kernel_primitive:NN \pdfrunninglinkoff     \tex_pdfrunninglinkoff:D
\__kernel_primitive:NN \pdfrunninglinkon      \tex_pdfrunninglinkon:D
\__kernel_primitive:NN \pdfsave               \tex_pdfsave:D
\__kernel_primitive:NN \pdfsetmatrix          \tex_pdfsetmatrix:D
\__kernel_primitive:NN \pdfstartlink          \tex_pdfstartlink:D
\__kernel_primitive:NN \pdfstartthread        \tex_pdfstartthread:D
\__kernel_primitive:NN \pdfsuppressptexinfo   \tex_pdfsuppressptexinfo:D
\__kernel_primitive:NN \pdfsuppresswarningdupdest
  \tex_pdfsuppresswarningdupdest:D
\__kernel_primitive:NN \pdfsuppresswarningdupmap
  \tex_pdfsuppresswarningdupmap:D
\__kernel_primitive:NN \pdfsuppresswarningpagegroup
  \tex_pdfsuppresswarningpagegroup:D
\__kernel_primitive:NN \pdfthread             \tex_pdfthread:D
\__kernel_primitive:NN \pdfthreadmargin       \tex_pdfthreadmargin:D
\__kernel_primitive:NN \pdftrailer            \tex_pdftrailer:D
\__kernel_primitive:NN \pdftrailerid          \tex_pdftrailerid:D
\__kernel_primitive:NN \pdfuniqueresname      \tex_pdfuniqueresname:D
\__kernel_primitive:NN \pdfvorigin            \tex_pdfvorigin:D
\__kernel_primitive:NN \pdfxform              \tex_pdfxform:D
\__kernel_primitive:NN \pdfxformname          \tex_pdfxformname:D
\__kernel_primitive:NN \pdfximage             \tex_pdfximage:D
\__kernel_primitive:NN \pdfximagebbox         \tex_pdfximagebbox:D
```

These are not related to PDF output and either already appear in other engines without the `\pdf` prefix, or might reasonably do so at some future stage. We therefore drop the leading `pdf` here.

```
\__kernel_primitive:NN \ifpdfabsdim           \tex_ifabsdim:D
\__kernel_primitive:NN \ifpdfabsnum           \tex_ifabsnum:D
\__kernel_primitive:NN \ifpdfprimitive        \tex_ifprimitive:D
\__kernel_primitive:NN \pdfadjustinterwordglue
  \tex_adjustinterwordglue:D
\__kernel_primitive:NN \pdfadjustspacing      \tex_adjustspacing:D
\__kernel_primitive:NN \pdfappendkern         \tex_appendkern:D
\__kernel_primitive:NN \pdfcopyfont           \tex_copyfont:D
```

```
\__kernel_primitive:NN \pdfcreationdate       \tex_creationdate:D
\__kernel_primitive:NN \pdfdraftmode          \tex_draftmode:D
\__kernel_primitive:NN \pdfeachlinedepth      \tex_eachlinedepth:D
\__kernel_primitive:NN \pdfeachlineheight     \tex_eachlineheight:D
\__kernel_primitive:NN \pdfelapsedtime        \tex_elapsedtime:D
\__kernel_primitive:NN \pdfescapehex          \tex_escapehex:D
\__kernel_primitive:NN \pdfescapename         \tex_escapename:D
\__kernel_primitive:NN \pdfescapestring       \tex_escapestring:D
\__kernel_primitive:NN \pdffirstlineheight    \tex_firstlineheight:D
\__kernel_primitive:NN \pdffontexpand         \tex_fontexpand:D
\__kernel_primitive:NN \pdffontsize           \tex_fontsize:D
\__kernel_primitive:NN \pdfignoreddimen       \tex_ignoreddimen:D
\__kernel_primitive:NN \pdfinsertht           \tex_insertht:D
\__kernel_primitive:NN \pdflastlinedepth      \tex_lastlinedepth:D
\__kernel_primitive:NN \pdflastmatch          \tex_lastmatch:D
\__kernel_primitive:NN \pdflastxpos           \tex_lastxpos:D
\__kernel_primitive:NN \pdflastypos           \tex_lastypos:D
\__kernel_primitive:NN \pdfmatch              \tex_match:D
\__kernel_primitive:NN \pdfnoligatures        \tex_noligatures:D
\__kernel_primitive:NN \pdfnormaldeviate      \tex_normaldeviate:D
\__kernel_primitive:NN \pdfpageheight         \tex_pageheight:D
\__kernel_primitive:NN \pdfpagewidth          \tex_pagewidth:D
\__kernel_primitive:NN \pdfpkmode             \tex_pkmode:D
\__kernel_primitive:NN \pdfpkresolution       \tex_pkresolution:D
\__kernel_primitive:NN \pdfprimitive          \tex_primitive:D
\__kernel_primitive:NN \pdfprependkern        \tex_prependkern:D
\__kernel_primitive:NN \pdfprotrudechars      \tex_protrudechars:D
\__kernel_primitive:NN \pdfpxdimen            \tex_pxdimen:D
\__kernel_primitive:NN \pdfrandomseed         \tex_randomseed:D
\__kernel_primitive:NN \pdfresettimer         \tex_resettimer:D
\__kernel_primitive:NN \pdfsavepos            \tex_savepos:D
\__kernel_primitive:NN \pdfsetrandomseed      \tex_setrandomseed:D
\__kernel_primitive:NN \pdfshellescape        \tex_shellescape:D
\__kernel_primitive:NN \pdftracingfonts       \tex_tracingfonts:D
\__kernel_primitive:NN \pdfunescapehex        \tex_unescapehex:D
\__kernel_primitive:NN \pdfuniformdeviate     \tex_uniformdeviate:D
```

The version primitives are not related to PDF mode but are pdfTEX-specific, so again are carried forward unchanged.

```
\__kernel_primitive:NN \pdftexbanner          \tex_pdftexbanner:D
\__kernel_primitive:NN \pdftexrevision        \tex_pdftexrevision:D
\__kernel_primitive:NN \pdftexversion         \tex_pdftexversion:D
```

These ones appear in pdfTEX but don't have `pdf` in the name at all: no decisions to make.

```
\__kernel_primitive:NN \efcode                \tex_efcode:D
\__kernel_primitive:NN \ifincsname            \tex_ifincsname:D
\__kernel_primitive:NN \knaccode              \tex_knaccode:D
\__kernel_primitive:NN \knbccode              \tex_knbccode:D
\__kernel_primitive:NN \knbscode              \tex_knbscode:D
\__kernel_primitive:NN \leftmarginkern        \tex_leftmarginkern:D
\__kernel_primitive:NN \letterspacefont       \tex_letterspacefont:D
\__kernel_primitive:NN \lpcode                \tex_lpcode:D
\__kernel_primitive:NN \quitvmode             \tex_quitvmode:D
\__kernel_primitive:NN \rightmarginkern       \tex_rightmarginkern:D
```

```
  \__kernel_primitive:NN \rpcode                \tex_rpcode:D
  \__kernel_primitive:NN \shbscode              \tex_shbscode:D
  \__kernel_primitive:NN \stbscode              \tex_stbscode:D
  \__kernel_primitive:NN \synctex               \tex_synctex:D
  \__kernel_primitive:NN \tagcode               \tex_tagcode:D
```

Post pdfTeX primitive availability gets more complex. Both XeTeX and LuaTeX have varying names for some primitives from pdfTeX. Particularly for LuaTeX tracking all of that would be hard. Instead, we now check that we only save primitives if they actually exist.

```
⟨/names | package⟩
⟨*package⟩
  \tex_long:D \tex_def:D \use_ii:nn #1#2 {#2}
  \tex_long:D \tex_def:D \use_none:n #1 { }
  \tex_long:D \tex_def:D \__kernel_primitive:NN #1#2
    {
      \tex_ifdefined:D #1
        \tex_expandafter:D \use_ii:nn
      \tex_fi:D
        \use_none:n { \tex_global:D \tex_let:D #2 #1 }
    }
⟨/package⟩
⟨*names | package⟩
```

Some pdfTeX primitives are handled here because they got dropped in LuaTeX but the corresponding internal names are emulated later. The Lua code is already loaded at this point, so we shouldn't overwrite them.

```
  \__kernel_primitive:NN \pdfstrcmp             \tex_strcmp:D
  \__kernel_primitive:NN \pdffilesize           \tex_filesize:D
  \__kernel_primitive:NN \pdfmdfivesum          \tex_mdfivesum:D
  \__kernel_primitive:NN \pdffilemoddate        \tex_filemoddate:D
  \__kernel_primitive:NN \pdffiledump           \tex_filedump:D
```

XeTeX-specific primitives. Note that XeTeX's `\strcmp` is handled earlier and is "rolled up" into `\pdfstrcmp`. A few cross-compatibility names which lack the `pdf` of the original are handled later.

```
  \__kernel_primitive:NN \suppressfontnotfounderror
    \tex_suppressfontnotfounderror:D
  \__kernel_primitive:NN \XeTeXcharclass        \tex_XeTeXcharclass:D
  \__kernel_primitive:NN \XeTeXcharglyph        \tex_XeTeXcharglyph:D
  \__kernel_primitive:NN \XeTeXcountfeatures    \tex_XeTeXcountfeatures:D
  \__kernel_primitive:NN \XeTeXcountglyphs      \tex_XeTeXcountglyphs:D
  \__kernel_primitive:NN \XeTeXcountselectors   \tex_XeTeXcountselectors:D
  \__kernel_primitive:NN \XeTeXcountvariations  \tex_XeTeXcountvariations:D
  \__kernel_primitive:NN \XeTeXdefaultencoding  \tex_XeTeXdefaultencoding:D
  \__kernel_primitive:NN \XeTeXdashbreakstate   \tex_XeTeXdashbreakstate:D
  \__kernel_primitive:NN \XeTeXfeaturecode      \tex_XeTeXfeaturecode:D
  \__kernel_primitive:NN \XeTeXfeaturename      \tex_XeTeXfeaturename:D
  \__kernel_primitive:NN \XeTeXfindfeaturebyname
    \tex_XeTeXfindfeaturebyname:D
  \__kernel_primitive:NN \XeTeXfindselectorbyname
    \tex_XeTeXfindselectorbyname:D
  \__kernel_primitive:NN \XeTeXfindvariationbyname
    \tex_XeTeXfindvariationbyname:D
  \__kernel_primitive:NN \XeTeXfirstfontchar    \tex_XeTeXfirstfontchar:D
```

```
\__kernel_primitive:NN \XeTeXfonttype         \tex_XeTeXfonttype:D
\__kernel_primitive:NN \XeTeXgenerateactualtext
  \tex_XeTeXgenerateactualtext:D
\__kernel_primitive:NN \XeTeXglyph            \tex_XeTeXglyph:D
\__kernel_primitive:NN \XeTeXglyphbounds      \tex_XeTeXglyphbounds:D
\__kernel_primitive:NN \XeTeXglyphindex       \tex_XeTeXglyphindex:D
\__kernel_primitive:NN \XeTeXglyphname        \tex_XeTeXglyphname:D
\__kernel_primitive:NN \XeTeXinputencoding    \tex_XeTeXinputencoding:D
\__kernel_primitive:NN \XeTeXinputnormalization
  \tex_XeTeXinputnormalization:D
\__kernel_primitive:NN \XeTeXinterchartokenstate
  \tex_XeTeXinterchartokenstate:D
\__kernel_primitive:NN \XeTeXinterchartoks    \tex_XeTeXinterchartoks:D
\__kernel_primitive:NN \XeTeXisdefaultselector
  \tex_XeTeXisdefaultselector:D
\__kernel_primitive:NN \XeTeXisexclusivefeature
  \tex_XeTeXisexclusivefeature:D
\__kernel_primitive:NN \XeTeXlastfontchar     \tex_XeTeXlastfontchar:D
\__kernel_primitive:NN \XeTeXlinebreakskip    \tex_XeTeXlinebreakskip:D
\__kernel_primitive:NN \XeTeXlinebreaklocale  \tex_XeTeXlinebreaklocale:D
\__kernel_primitive:NN \XeTeXlinebreakpenalty \tex_XeTeXlinebreakpenalty:D
\__kernel_primitive:NN \XeTeXOTcountfeatures  \tex_XeTeXOTcountfeatures:D
\__kernel_primitive:NN \XeTeXOTcountlanguages \tex_XeTeXOTcountlanguages:D
\__kernel_primitive:NN \XeTeXOTcountscripts   \tex_XeTeXOTcountscripts:D
\__kernel_primitive:NN \XeTeXOTfeaturetag     \tex_XeTeXOTfeaturetag:D
\__kernel_primitive:NN \XeTeXOTlanguagetag    \tex_XeTeXOTlanguagetag:D
\__kernel_primitive:NN \XeTeXOTscripttag      \tex_XeTeXOTscripttag:D
\__kernel_primitive:NN \XeTeXpdffile          \tex_XeTeXpdffile:D
\__kernel_primitive:NN \XeTeXpdfpagecount     \tex_XeTeXpdfpagecount:D
\__kernel_primitive:NN \XeTeXpicfile          \tex_XeTeXpicfile:D
\__kernel_primitive:NN \XeTeXrevision         \tex_XeTeXrevision:D
\__kernel_primitive:NN \XeTeXselectorname     \tex_XeTeXselectorname:D
\__kernel_primitive:NN \XeTeXtracingfonts     \tex_XeTeXtracingfonts:D
\__kernel_primitive:NN \XeTeXupwardsmode      \tex_XeTeXupwardsmode:D
\__kernel_primitive:NN \XeTeXuseglyphmetrics  \tex_XeTeXuseglyphmetrics:D
\__kernel_primitive:NN \XeTeXvariation        \tex_XeTeXvariation:D
\__kernel_primitive:NN \XeTeXvariationdefault \tex_XeTeXvariationdefault:D
\__kernel_primitive:NN \XeTeXvariationmax     \tex_XeTeXvariationmax:D
\__kernel_primitive:NN \XeTeXvariationmin     \tex_XeTeXvariationmin:D
\__kernel_primitive:NN \XeTeXvariationname    \tex_XeTeXvariationname:D
\__kernel_primitive:NN \XeTeXversion          \tex_XeTeXversion:D
\__kernel_primitive:NN \XeTeXselectorcode     \tex_XeTeXselectorcode:D
\__kernel_primitive:NN \XeTeXinterwordspaceshaping
                 \tex_XeTeXinterwordspaceshaping:D
\__kernel_primitive:NN \XeTeXhyphenatablelength
                 \tex_XeTeXhyphenatablelength:D
```

Primitives from pdfTEX that XƎTEX renames: also helps with LuaTEX.

```
\__kernel_primitive:NN \creationdate          \tex_creationdate:D
\__kernel_primitive:NN \elapsedtime           \tex_elapsedtime:D
\__kernel_primitive:NN \filedump              \tex_filedump:D
\__kernel_primitive:NN \filemoddate           \tex_filemoddate:D
\__kernel_primitive:NN \filesize              \tex_filesize:D
\__kernel_primitive:NN \mdfivesum             \tex_mdfivesum:D
\__kernel_primitive:NN \ifprimitive           \tex_ifprimitive:D
```

```
\__kernel_primitive:NN \primitive             \tex_primitive:D
\__kernel_primitive:NN \resettimer            \tex_resettimer:D
\__kernel_primitive:NN \shellescape           \tex_shellescape:D
\__kernel_primitive:NN \XeTeXprotrudechars    \tex_protrudechars:D
```

Primitives from LuaTeX, some of which have been ported back to XeTeX.

```
\__kernel_primitive:NN \alignmark             \tex_alignmark:D
\__kernel_primitive:NN \aligntab              \tex_aligntab:D
\__kernel_primitive:NN \attribute             \tex_attribute:D
\__kernel_primitive:NN \attributedef          \tex_attributedef:D
\__kernel_primitive:NN \automaticdiscretionary
  \tex_automaticdiscretionary:D
\__kernel_primitive:NN \automatichyphenmode   \tex_automatichyphenmode:D
\__kernel_primitive:NN \automatichyphenpenalty
  \tex_automatichyphenpenalty:D
\__kernel_primitive:NN \begincsname           \tex_begincsname:D
\__kernel_primitive:NN \bodydir               \tex_bodydir:D
\__kernel_primitive:NN \bodydirection         \tex_bodydirection:D
\__kernel_primitive:NN \boundary              \tex_boundary:D
\__kernel_primitive:NN \boxdir                \tex_boxdir:D
\__kernel_primitive:NN \boxdirection          \tex_boxdirection:D
\__kernel_primitive:NN \breakafterdirmode     \tex_breakafterdirmode:D
\__kernel_primitive:NN \catcodetable          \tex_catcodetable:D
\__kernel_primitive:NN \clearmarks            \tex_clearmarks:D
% \__kernel_primitive:NN \compoundhyphenmode
%   \tex_compoundhyphenmode:D % not documented in manual
\__kernel_primitive:NN \crampeddisplaystyle   \tex_crampeddisplaystyle:D
\__kernel_primitive:NN \crampedscriptscriptstyle
  \tex_crampedscriptscriptstyle:D
\__kernel_primitive:NN \crampedscriptstyle    \tex_crampedscriptstyle:D
\__kernel_primitive:NN \crampedtextstyle      \tex_crampedtextstyle:D
\__kernel_primitive:NN \csstring              \tex_csstring:D
\__kernel_primitive:NN \deferred              \tex_deferred:D
\__kernel_primitive:NN \discretionaryligaturemode
  \tex_discretionaryligaturemode:D
\__kernel_primitive:NN \directlua             \tex_directlua:D
\__kernel_primitive:NN \dviextension          \tex_dviextension:D
\__kernel_primitive:NN \dvifeedback           \tex_dvifeedback:D
\__kernel_primitive:NN \dvivariable           \tex_dvivariable:D
\__kernel_primitive:NN \eTeXglueshrinkorder   \tex_eTeXglueshrinkorder:D
\__kernel_primitive:NN \eTeXgluestretchorder  \tex_eTeXgluestretchorder:D
\__kernel_primitive:NN \endlocalcontrol       \tex_endlocalcontrol:D
\__kernel_primitive:NN \etoksapp              \tex_etoksapp:D
\__kernel_primitive:NN \etokspre              \tex_etokspre:D
\__kernel_primitive:NN \exceptionpenalty      \tex_exceptionpenalty:D
\__kernel_primitive:NN \exhyphenchar          \tex_exhyphenchar:D
\__kernel_primitive:NN \explicithyphenpenalty \tex_explicithyphenpenalty:D
\__kernel_primitive:NN \expanded              \tex_expanded:D
\__kernel_primitive:NN \explicitdiscretionary \tex_explicitdiscretionary:D
\__kernel_primitive:NN \firstvalidlanguage    \tex_firstvalidlanguage:D
% \__kernel_primitive:NN \fixupboxesmode
%   \tex_fixupboxesmode:D % not documented in manual
\__kernel_primitive:NN \fontid                \tex_fontid:D
\__kernel_primitive:NN \formatname            \tex_formatname:D
\__kernel_primitive:NN \hjcode                \tex_hjcode:D
```

```
  \__kernel_primitive:NN \hpack                  \tex_hpack:D
  \__kernel_primitive:NN \hyphenationbounds      \tex_hyphenationbounds:D
  \__kernel_primitive:NN \hyphenationmin         \tex_hyphenationmin:D
  \__kernel_primitive:NN \hyphenpenaltymode      \tex_hyphenpenaltymode:D
  \__kernel_primitive:NN \gleaders               \tex_gleaders:D
  \__kernel_primitive:NN \glet                   \tex_glet:D
  \__kernel_primitive:NN \glyphdimensionsmode    \tex_glyphdimensionsmode:D
  \__kernel_primitive:NN \gtoksapp               \tex_gtoksapp:D
  \__kernel_primitive:NN \gtokspre               \tex_gtokspre:D
  \__kernel_primitive:NN \ifcondition            \tex_ifcondition:D
  \__kernel_primitive:NN \immediateassigned      \tex_immediateassigned:D
  \__kernel_primitive:NN \immediateassignment    \tex_immediateassignment:D
  \__kernel_primitive:NN \initcatcodetable       \tex_initcatcodetable:D
  \__kernel_primitive:NN \lastnamedcs            \tex_lastnamedcs:D
  \__kernel_primitive:NN \latelua                \tex_latelua:D
  \__kernel_primitive:NN \lateluafunction        \tex_lateluafunction:D
  \__kernel_primitive:NN \leftghost              \tex_leftghost:D
  \__kernel_primitive:NN \letcharcode            \tex_letcharcode:D
  \__kernel_primitive:NN \linedir                \tex_linedir:D
  \__kernel_primitive:NN \linedirection          \tex_linedirection:D
  \__kernel_primitive:NN \localbrokenpenalty     \tex_localbrokenpenalty:D
  \__kernel_primitive:NN \localinterlinepenalty  \tex_localinterlinepenalty:D
  \__kernel_primitive:NN \luabytecode            \tex_luabytecode:D
  \__kernel_primitive:NN \luabytecodecall        \tex_luabytecodecall:D
  \__kernel_primitive:NN \luacopyinputnodes      \tex_luacopyinputnodes:D
  \__kernel_primitive:NN \luadef                 \tex_luadef:D
  \__kernel_primitive:NN \localleftbox           \tex_localleftbox:D
  \__kernel_primitive:NN \localrightbox          \tex_localrightbox:D
  \__kernel_primitive:NN \luaescapestring        \tex_luaescapestring:D
  \__kernel_primitive:NN \luafunction            \tex_luafunction:D
  \__kernel_primitive:NN \luafunctioncall        \tex_luafunctioncall:D
  \__kernel_primitive:NN \luatexbanner           \tex_luatexbanner:D
  \__kernel_primitive:NN \luatexrevision         \tex_luatexrevision:D
  \__kernel_primitive:NN \luatexversion          \tex_luatexversion:D
  \__kernel_primitive:NN \mathdefaultsmode       \tex_mathdefaultsmode:D
  \__kernel_primitive:NN \mathdelimitersmode     \tex_mathdelimitersmode:D
  \__kernel_primitive:NN \mathdir                \tex_mathdir:D
  \__kernel_primitive:NN \mathdirection          \tex_mathdirection:D
  \__kernel_primitive:NN \mathdisplayskipmode    \tex_mathdisplayskipmode:D
  \__kernel_primitive:NN \matheqdirmode          \tex_matheqdirmode:D
  \__kernel_primitive:NN \matheqnogapstep        \tex_matheqnogapstep:D
  \__kernel_primitive:NN \mathflattenmode        \tex_mathflattenmode:D
  \__kernel_primitive:NN \mathitalicsmode        \tex_mathitalicsmode:D
  \__kernel_primitive:NN \mathnolimitsmode       \tex_mathnolimitsmode:D
  \__kernel_primitive:NN \mathoption             \tex_mathoption:D
  \__kernel_primitive:NN \mathpenaltiesmode      \tex_mathpenaltiesmode:D
  \__kernel_primitive:NN \mathrulesfam           \tex_mathrulesfam:D
% \__kernel_primitive:NN \mathrulesmode
%   \tex_mathrulesmode:D % not documented in manual
% \__kernel_primitive:NN \mathrulethicknessmode
%   \tex_mathrulethicknessmode:D % not documented in manual
  \__kernel_primitive:NN \mathscriptsmode        \tex_mathscriptsmode:D
  \__kernel_primitive:NN \mathscriptboxmode      \tex_mathscriptboxmode:D
  \__kernel_primitive:NN \mathscriptcharmode     \tex_mathscriptcharmode:D
```

```
\__kernel_primitive:NN \mathstyle             \tex_mathstyle:D
\__kernel_primitive:NN \mathsurroundmode      \tex_mathsurroundmode:D
\__kernel_primitive:NN \mathsurroundskip      \tex_mathsurroundskip:D
\__kernel_primitive:NN \nohrule               \tex_nohrule:D
\__kernel_primitive:NN \nokerns               \tex_nokerns:D
\__kernel_primitive:NN \noligs                \tex_noligs:D
\__kernel_primitive:NN \nospaces              \tex_nospaces:D
\__kernel_primitive:NN \novrule               \tex_novrule:D
\__kernel_primitive:NN \outputbox             \tex_outputbox:D
\__kernel_primitive:NN \pagebottomoffset      \tex_pagebottomoffset:D
\__kernel_primitive:NN \pagedir               \tex_pagedir:D
\__kernel_primitive:NN \pagedirection         \tex_pagedirection:D
\__kernel_primitive:NN \pageleftoffset        \tex_pageleftoffset:D
\__kernel_primitive:NN \pagerightoffset       \tex_pagerightoffset:D
\__kernel_primitive:NN \pagetopoffset         \tex_pagetopoffset:D
\__kernel_primitive:NN \pardir                \tex_pardir:D
\__kernel_primitive:NN \pardirection          \tex_pardirection:D
\__kernel_primitive:NN \pdfextension          \tex_pdfextension:D
\__kernel_primitive:NN \pdffeedback           \tex_pdffeedback:D
\__kernel_primitive:NN \pdfvariable           \tex_pdfvariable:D
\__kernel_primitive:NN \postexhyphenchar      \tex_postexhyphenchar:D
\__kernel_primitive:NN \posthyphenchar        \tex_posthyphenchar:D
\__kernel_primitive:NN \prebinoppenalty       \tex_prebinoppenalty:D
\__kernel_primitive:NN \predisplaygapfactor   \tex_predisplaygapfactor:D
\__kernel_primitive:NN \preexhyphenchar       \tex_preexhyphenchar:D
\__kernel_primitive:NN \prehyphenchar         \tex_prehyphenchar:D
\__kernel_primitive:NN \prerelpenalty         \tex_prerelpenalty:D
\__kernel_primitive:NN \protrusionboundary    \tex_protrusionboundary:D
\__kernel_primitive:NN \rightghost            \tex_rightghost:D
\__kernel_primitive:NN \savecatcodetable      \tex_savecatcodetable:D
\__kernel_primitive:NN \scantextokens         \tex_scantextokens:D
\__kernel_primitive:NN \setfontid             \tex_setfontid:D
\__kernel_primitive:NN \shapemode             \tex_shapemode:D
\__kernel_primitive:NN \suppressifcsnameerror \tex_suppressifcsnameerror:D
\__kernel_primitive:NN \suppresslongerror     \tex_suppresslongerror:D
\__kernel_primitive:NN \suppressmathparerror  \tex_suppressmathparerror:D
\__kernel_primitive:NN \suppressoutererror    \tex_suppressoutererror:D
\__kernel_primitive:NN \suppressprimitiveerror
  \tex_suppressprimitiveerror:D
\__kernel_primitive:NN \textdir               \tex_textdir:D
\__kernel_primitive:NN \textdirection         \tex_textdirection:D
\__kernel_primitive:NN \toksapp               \tex_toksapp:D
\__kernel_primitive:NN \tokspre               \tex_tokspre:D
\__kernel_primitive:NN \tpack                 \tex_tpack:D
\__kernel_primitive:NN \variablefam           \tex_variablefam:D
\__kernel_primitive:NN \vpack                 \tex_vpack:D
\__kernel_primitive:NN \wordboundary          \tex_wordboundary:D
\__kernel_primitive:NN \xtoksapp              \tex_xtoksapp:D
\__kernel_primitive:NN \xtokspre              \tex_xtokspre:D
```

Primitives from pdfTeX that LuaTeX renames.

```
\__kernel_primitive:NN \adjustspacing         \tex_adjustspacing:D
\__kernel_primitive:NN \copyfont              \tex_copyfont:D
\__kernel_primitive:NN \draftmode             \tex_draftmode:D
\__kernel_primitive:NN \expandglyphsinfont    \tex_fontexpand:D
```

```
  \__kernel_primitive:NN \ifabsdim              \tex_ifabsdim:D
  \__kernel_primitive:NN \ifabsnum              \tex_ifabsnum:D
  \__kernel_primitive:NN \ignoreligaturesinfont \tex_ignoreligaturesinfont:D
  \__kernel_primitive:NN \insertht              \tex_insertht:D
  \__kernel_primitive:NN \lastsavedboxresourceindex
    \tex_pdflastxform:D
  \__kernel_primitive:NN \lastsavedimageresourceindex
    \tex_pdflastximage:D
  \__kernel_primitive:NN \lastsavedimageresourcepages
    \tex_pdflastximagepages:D
  \__kernel_primitive:NN \lastxpos              \tex_lastxpos:D
  \__kernel_primitive:NN \lastypos              \tex_lastypos:D
  \__kernel_primitive:NN \normaldeviate         \tex_normaldeviate:D
  \__kernel_primitive:NN \outputmode            \tex_pdfoutput:D
  \__kernel_primitive:NN \pageheight            \tex_pageheight:D
  \__kernel_primitive:NN \pagewidth             \tex_pagewidth:D
  \__kernel_primitive:NN \protrudechars         \tex_protrudechars:D
  \__kernel_primitive:NN \pxdimen               \tex_pxdimen:D
  \__kernel_primitive:NN \randomseed            \tex_randomseed:D
  \__kernel_primitive:NN \useboxresource        \tex_pdfrefxform:D
  \__kernel_primitive:NN \useimageresource      \tex_pdfrefximage:D
  \__kernel_primitive:NN \savepos               \tex_savepos:D
  \__kernel_primitive:NN \saveboxresource       \tex_pdfxform:D
  \__kernel_primitive:NN \saveimageresource     \tex_pdfximage:D
  \__kernel_primitive:NN \setrandomseed         \tex_setrandomseed:D
  \__kernel_primitive:NN \tracingfonts          \tex_tracingfonts:D
  \__kernel_primitive:NN \uniformdeviate        \tex_uniformdeviate:D
```

The set of Unicode math primitives were introduced by XeTeX and LuaTeX in a somewhat complex fashion: a few first as `\XeTeX...` which were then renamed with LuaTeX having a lot more. These names now all start `\U...` and mainly `\Umath...`.

```
  \__kernel_primitive:NN \Uchar                 \tex_Uchar:D
  \__kernel_primitive:NN \Ucharcat              \tex_Ucharcat:D
  \__kernel_primitive:NN \Udelcode              \tex_Udelcode:D
  \__kernel_primitive:NN \Udelcodenum           \tex_Udelcodenum:D
  \__kernel_primitive:NN \Udelimiter            \tex_Udelimiter:D
  \__kernel_primitive:NN \Udelimiterover        \tex_Udelimiterover:D
  \__kernel_primitive:NN \Udelimiterunder       \tex_Udelimiterunder:D
  \__kernel_primitive:NN \Uhextensible          \tex_Uhextensible:D
  \__kernel_primitive:NN \Uleft                 \tex_Uleft:D
  \__kernel_primitive:NN \Umathaccent           \tex_Umathaccent:D
  \__kernel_primitive:NN \Umathaxis             \tex_Umathaxis:D
  \__kernel_primitive:NN \Umathbinbinspacing    \tex_Umathbinbinspacing:D
  \__kernel_primitive:NN \Umathbinclosespacing  \tex_Umathbinclosespacing:D
  \__kernel_primitive:NN \Umathbininnerspacing  \tex_Umathbininnerspacing:D
  \__kernel_primitive:NN \Umathbinopenspacing   \tex_Umathbinopenspacing:D
  \__kernel_primitive:NN \Umathbinopspacing     \tex_Umathbinopspacing:D
  \__kernel_primitive:NN \Umathbinordspacing    \tex_Umathbinordspacing:D
  \__kernel_primitive:NN \Umathbinpunctspacing  \tex_Umathbinpunctspacing:D
  \__kernel_primitive:NN \Umathbinrelspacing    \tex_Umathbinrelspacing:D
  \__kernel_primitive:NN \Umathchar             \tex_Umathchar:D
  \__kernel_primitive:NN \Umathcharclass        \tex_Umathcharclass:D
  \__kernel_primitive:NN \Umathchardef          \tex_Umathchardef:D
  \__kernel_primitive:NN \Umathcharfam          \tex_Umathcharfam:D
```

```
\__kernel_primitive:NN \Umathcharnum          \tex_Umathcharnum:D
\__kernel_primitive:NN \Umathcharnumdef       \tex_Umathcharnumdef:D
\__kernel_primitive:NN \Umathcharslot         \tex_Umathcharslot:D
\__kernel_primitive:NN \Umathclosebinspacing  \tex_Umathclosebinspacing:D
\__kernel_primitive:NN \Umathcloseclosespacing
  \tex_Umathcloseclosespacing:D
\__kernel_primitive:NN \Umathcloseinnerspacing
  \tex_Umathcloseinnerspacing:D
\__kernel_primitive:NN \Umathcloseopenspacing \tex_Umathcloseopenspacing:D
\__kernel_primitive:NN \Umathcloseopspacing   \tex_Umathcloseopspacing:D
\__kernel_primitive:NN \Umathcloseordspacing  \tex_Umathcloseordspacing:D
\__kernel_primitive:NN \Umathclosepunctspacing
  \tex_Umathclosepunctspacing:D
\__kernel_primitive:NN \Umathcloserelspacing  \tex_Umathcloserelspacing:D
\__kernel_primitive:NN \Umathcode             \tex_Umathcode:D
\__kernel_primitive:NN \Umathcodenum          \tex_Umathcodenum:D
\__kernel_primitive:NN \Umathconnectoroverlapmin
  \tex_Umathconnectoroverlapmin:D
\__kernel_primitive:NN \Umathfractiondelsize  \tex_Umathfractiondelsize:D
\__kernel_primitive:NN \Umathfractiondenomdown
  \tex_Umathfractiondenomdown:D
\__kernel_primitive:NN \Umathfractiondenomvgap
  \tex_Umathfractiondenomvgap:D
\__kernel_primitive:NN \Umathfractionnumup    \tex_Umathfractionnumup:D
\__kernel_primitive:NN \Umathfractionnumvgap  \tex_Umathfractionnumvgap:D
\__kernel_primitive:NN \Umathfractionrule     \tex_Umathfractionrule:D
\__kernel_primitive:NN \Umathinnerbinspacing  \tex_Umathinnerbinspacing:D
\__kernel_primitive:NN \Umathinnerclosespacing
  \tex_Umathinnerclosespacing:D
\__kernel_primitive:NN \Umathinnerinnerspacing
  \tex_Umathinnerinnerspacing:D
\__kernel_primitive:NN \Umathinneropenspacing \tex_Umathinneropenspacing:D
\__kernel_primitive:NN \Umathinneropspacing   \tex_Umathinneropspacing:D
\__kernel_primitive:NN \Umathinnerordspacing  \tex_Umathinnerordspacing:D
\__kernel_primitive:NN \Umathinnerpunctspacing
  \tex_Umathinnerpunctspacing:D
\__kernel_primitive:NN \Umathinnerrelspacing  \tex_Umathinnerrelspacing:D
\__kernel_primitive:NN \Umathlimitabovebgap   \tex_Umathlimitabovebgap:D
\__kernel_primitive:NN \Umathlimitabovekern   \tex_Umathlimitabovekern:D
\__kernel_primitive:NN \Umathlimitabovevgap   \tex_Umathlimitabovevgap:D
\__kernel_primitive:NN \Umathlimitbelowbgap   \tex_Umathlimitbelowbgap:D
\__kernel_primitive:NN \Umathlimitbelowkern   \tex_Umathlimitbelowkern:D
\__kernel_primitive:NN \Umathlimitbelowvgap   \tex_Umathlimitbelowvgap:D
\__kernel_primitive:NN \Umathnolimitsubfactor \tex_Umathnolimitsubfactor:D
\__kernel_primitive:NN \Umathnolimitsupfactor \tex_Umathnolimitsupfactor:D
\__kernel_primitive:NN \Umathopbinspacing     \tex_Umathopbinspacing:D
\__kernel_primitive:NN \Umathopclosespacing   \tex_Umathopclosespacing:D
\__kernel_primitive:NN \Umathopenbinspacing   \tex_Umathopenbinspacing:D
\__kernel_primitive:NN \Umathopenclosespacing \tex_Umathopenclosespacing:D
\__kernel_primitive:NN \Umathopeninnerspacing \tex_Umathopeninnerspacing:D
\__kernel_primitive:NN \Umathopenopenspacing  \tex_Umathopenopenspacing:D
\__kernel_primitive:NN \Umathopenopspacing    \tex_Umathopenopspacing:D
\__kernel_primitive:NN \Umathopenordspacing   \tex_Umathopenordspacing:D
\__kernel_primitive:NN \Umathopenpunctspacing \tex_Umathopenpunctspacing:D
```

```
\__kernel_primitive:NN \Umathopenrelspacing   \tex_Umathopenrelspacing:D
\__kernel_primitive:NN \Umathoperatorsize     \tex_Umathoperatorsize:D
\__kernel_primitive:NN \Umathopinnerspacing   \tex_Umathopinnerspacing:D
\__kernel_primitive:NN \Umathopopenspacing    \tex_Umathopopenspacing:D
\__kernel_primitive:NN \Umathopopspacing      \tex_Umathopopspacing:D
\__kernel_primitive:NN \Umathopordspacing     \tex_Umathopordspacing:D
\__kernel_primitive:NN \Umathoppunctspacing   \tex_Umathoppunctspacing:D
\__kernel_primitive:NN \Umathoprelspacing     \tex_Umathoprelspacing:D
\__kernel_primitive:NN \Umathordbinspacing    \tex_Umathordbinspacing:D
\__kernel_primitive:NN \Umathordclosespacing  \tex_Umathordclosespacing:D
\__kernel_primitive:NN \Umathordinnerspacing  \tex_Umathordinnerspacing:D
\__kernel_primitive:NN \Umathordopenspacing   \tex_Umathordopenspacing:D
\__kernel_primitive:NN \Umathordopspacing     \tex_Umathordopspacing:D
\__kernel_primitive:NN \Umathordordspacing    \tex_Umathordordspacing:D
\__kernel_primitive:NN \Umathordpunctspacing  \tex_Umathordpunctspacing:D
\__kernel_primitive:NN \Umathordrelspacing    \tex_Umathordrelspacing:D
\__kernel_primitive:NN \Umathoverbarkern      \tex_Umathoverbarkern:D
\__kernel_primitive:NN \Umathoverbarrule      \tex_Umathoverbarrule:D
\__kernel_primitive:NN \Umathoverbarvgap      \tex_Umathoverbarvgap:D
\__kernel_primitive:NN \Umathoverdelimiterbgap
  \tex_Umathoverdelimiterbgap:D
\__kernel_primitive:NN \Umathoverdelimitervgap
  \tex_Umathoverdelimitervgap:D
\__kernel_primitive:NN \Umathpunctbinspacing  \tex_Umathpunctbinspacing:D
\__kernel_primitive:NN \Umathpunctclosespacing
  \tex_Umathpunctclosespacing:D
\__kernel_primitive:NN \Umathpunctinnerspacing
  \tex_Umathpunctinnerspacing:D
\__kernel_primitive:NN \Umathpunctopenspacing \tex_Umathpunctopenspacing:D
\__kernel_primitive:NN \Umathpunctopspacing   \tex_Umathpunctopspacing:D
\__kernel_primitive:NN \Umathpunctordspacing  \tex_Umathpunctordspacing:D
\__kernel_primitive:NN \Umathpunctpunctspacing
  \tex_Umathpunctpunctspacing:D
\__kernel_primitive:NN \Umathpunctrelspacing  \tex_Umathpunctrelspacing:D
\__kernel_primitive:NN \Umathquad             \tex_Umathquad:D
\__kernel_primitive:NN \Umathradicaldegreeafter
  \tex_Umathradicaldegreeafter:D
\__kernel_primitive:NN \Umathradicaldegreebefore
  \tex_Umathradicaldegreebefore:D
\__kernel_primitive:NN \Umathradicaldegreeraise
  \tex_Umathradicaldegreeraise:D
\__kernel_primitive:NN \Umathradicalkern      \tex_Umathradicalkern:D
\__kernel_primitive:NN \Umathradicalrule      \tex_Umathradicalrule:D
\__kernel_primitive:NN \Umathradicalvgap      \tex_Umathradicalvgap:D
\__kernel_primitive:NN \Umathrelbinspacing    \tex_Umathrelbinspacing:D
\__kernel_primitive:NN \Umathrelclosespacing  \tex_Umathrelclosespacing:D
\__kernel_primitive:NN \Umathrelinnerspacing  \tex_Umathrelinnerspacing:D
\__kernel_primitive:NN \Umathrelopenspacing   \tex_Umathrelopenspacing:D
\__kernel_primitive:NN \Umathrelopspacing     \tex_Umathrelopspacing:D
\__kernel_primitive:NN \Umathrelordspacing    \tex_Umathrelordspacing:D
\__kernel_primitive:NN \Umathrelpunctspacing  \tex_Umathrelpunctspacing:D
\__kernel_primitive:NN \Umathrelrelspacing    \tex_Umathrelrelspacing:D
\__kernel_primitive:NN \Umathskewedfractionhgap
  \tex_Umathskewedfractionhgap:D
```

```
\__kernel_primitive:NN \Umathskewedfractionvgap
  \tex_Umathskewedfractionvgap:D
\__kernel_primitive:NN \Umathspaceafterscript \tex_Umathspaceafterscript:D
\__kernel_primitive:NN \Umathstackdenomdown   \tex_Umathstackdenomdown:D
\__kernel_primitive:NN \Umathstacknumup       \tex_Umathstacknumup:D
\__kernel_primitive:NN \Umathstackvgap        \tex_Umathstackvgap:D
\__kernel_primitive:NN \Umathsubshiftdown     \tex_Umathsubshiftdown:D
\__kernel_primitive:NN \Umathsubshiftdrop     \tex_Umathsubshiftdrop:D
\__kernel_primitive:NN \Umathsubsupshiftdown  \tex_Umathsubsupshiftdown:D
\__kernel_primitive:NN \Umathsubsupvgap       \tex_Umathsubsupvgap:D
\__kernel_primitive:NN \Umathsubtopmax        \tex_Umathsubtopmax:D
\__kernel_primitive:NN \Umathsupbottommin     \tex_Umathsupbottommin:D
\__kernel_primitive:NN \Umathsupshiftdrop     \tex_Umathsupshiftdrop:D
\__kernel_primitive:NN \Umathsupshiftup       \tex_Umathsupshiftup:D
\__kernel_primitive:NN \Umathsupsubbottommax  \tex_Umathsupsubbottommax:D
\__kernel_primitive:NN \Umathunderbarkern     \tex_Umathunderbarkern:D
\__kernel_primitive:NN \Umathunderbarrule     \tex_Umathunderbarrule:D
\__kernel_primitive:NN \Umathunderbarvgap     \tex_Umathunderbarvgap:D
\__kernel_primitive:NN \Umathunderdelimiterbgap
  \tex_Umathunderdelimiterbgap:D
\__kernel_primitive:NN \Umathunderdelimitervgap
  \tex_Umathunderdelimitervgap:D
\__kernel_primitive:NN \Umiddle               \tex_Umiddle:D
\__kernel_primitive:NN \Unosubscript          \tex_Unosubscript:D
\__kernel_primitive:NN \Unosuperscript        \tex_Unosuperscript:D
\__kernel_primitive:NN \Uoverdelimiter        \tex_Uoverdelimiter:D
\__kernel_primitive:NN \Uradical              \tex_Uradical:D
\__kernel_primitive:NN \Uright                \tex_Uright:D
\__kernel_primitive:NN \Uroot                 \tex_Uroot:D
\__kernel_primitive:NN \Uskewed               \tex_Uskewed:D
\__kernel_primitive:NN \Uskewedwithdelims     \tex_Uskewedwithdelims:D
\__kernel_primitive:NN \Ustack                \tex_Ustack:D
\__kernel_primitive:NN \Ustartdisplaymath     \tex_Ustartdisplaymath:D
\__kernel_primitive:NN \Ustartmath            \tex_Ustartmath:D
\__kernel_primitive:NN \Ustopdisplaymath      \tex_Ustopdisplaymath:D
\__kernel_primitive:NN \Ustopmath             \tex_Ustopmath:D
\__kernel_primitive:NN \Usubscript            \tex_Usubscript:D
\__kernel_primitive:NN \Usuperscript          \tex_Usuperscript:D
\__kernel_primitive:NN \Uunderdelimiter       \tex_Uunderdelimiter:D
\__kernel_primitive:NN \Uvextensible          \tex_Uvextensible:D
```

Primitives from pTEX.

```
\__kernel_primitive:NN \autospacing           \tex_autospacing:D
\__kernel_primitive:NN \autoxspacing          \tex_autoxspacing:D
\__kernel_primitive:NN \currentcjktoken       \tex_currentcjktoken:D
\__kernel_primitive:NN \currentspacingmode    \tex_currentspacingmode:D
\__kernel_primitive:NN \currentxspacingmode   \tex_currentxspacingmode:D
\__kernel_primitive:NN \disinhibitglue        \tex_disinhibitglue:D
\__kernel_primitive:NN \dtou                  \tex_dtou:D
\__kernel_primitive:NN \epTeXinputencoding    \tex_epTeXinputencoding:D
\__kernel_primitive:NN \epTeXversion          \tex_epTeXversion:D
\__kernel_primitive:NN \euc                   \tex_euc:D
\__kernel_primitive:NN \hfi                   \tex_hfi:D
\__kernel_primitive:NN \ifdbox                \tex_ifdbox:D
\__kernel_primitive:NN \ifddir                \tex_ifddir:D
```

```
\__kernel_primitive:NN \ifjfont               \tex_ifjfont:D
\__kernel_primitive:NN \ifmbox                \tex_ifmbox:D
\__kernel_primitive:NN \ifmdir                \tex_ifmdir:D
\__kernel_primitive:NN \iftbox                \tex_iftbox:D
\__kernel_primitive:NN \iftfont               \tex_iftfont:D
\__kernel_primitive:NN \iftdir                \tex_iftdir:D
\__kernel_primitive:NN \ifybox                \tex_ifybox:D
\__kernel_primitive:NN \ifydir                \tex_ifydir:D
\__kernel_primitive:NN \inhibitglue           \tex_inhibitglue:D
\__kernel_primitive:NN \inhibitxspcode        \tex_inhibitxspcode:D
\__kernel_primitive:NN \jcharwidowpenalty     \tex_jcharwidowpenalty:D
\__kernel_primitive:NN \jfam                  \tex_jfam:D
\__kernel_primitive:NN \jfont                 \tex_jfont:D
\__kernel_primitive:NN \jis                   \tex_jis:D
\__kernel_primitive:NN \kanjiskip             \tex_kanjiskip:D
\__kernel_primitive:NN \kansuji               \tex_kansuji:D
\__kernel_primitive:NN \kansujichar           \tex_kansujichar:D
\__kernel_primitive:NN \kcatcode              \tex_kcatcode:D
\__kernel_primitive:NN \kuten                 \tex_kuten:D
\__kernel_primitive:NN \lastnodechar          \tex_lastnodechar:D
\__kernel_primitive:NN \lastnodefont          \tex_lastnodefont:D
\__kernel_primitive:NN \lastnodesubtype       \tex_lastnodesubtype:D
\__kernel_primitive:NN \noautospacing         \tex_noautospacing:D
\__kernel_primitive:NN \noautoxspacing        \tex_noautoxspacing:D
\__kernel_primitive:NN \pagefistretch         \tex_pagefistretch:D
\__kernel_primitive:NN \postbreakpenalty      \tex_postbreakpenalty:D
\__kernel_primitive:NN \prebreakpenalty       \tex_prebreakpenalty:D
\__kernel_primitive:NN \ptexfontname          \tex_ptexfontname:D
\__kernel_primitive:NN \ptexlineendmode       \tex_lineendmode:D
\__kernel_primitive:NN \ptexminorversion      \tex_ptexminorversion:D
\__kernel_primitive:NN \ptexrevision          \tex_ptexrevision:D
\__kernel_primitive:NN \ptextracingfonts      \tex_ptextracingfonts:D
\__kernel_primitive:NN \ptexversion           \tex_ptexversion:D
\__kernel_primitive:NN \readpapersizespecial  \tex_readpapersizespecial:D
\__kernel_primitive:NN \scriptbaselineshiftfactor
  \tex_scriptbaselineshiftfactor:D
\__kernel_primitive:NN \scriptscriptbaselineshiftfactor
  \tex_scriptscriptbaselineshiftfactor:D
\__kernel_primitive:NN \showmode              \tex_showmode:D
\__kernel_primitive:NN \sjis                  \tex_sjis:D
\__kernel_primitive:NN \tate                  \tex_tate:D
\__kernel_primitive:NN \tbaselineshift        \tex_tbaselineshift:D
\__kernel_primitive:NN \textbaselineshiftfactor
  \tex_textbaselineshiftfactor:D
\__kernel_primitive:NN \tfont                 \tex_tfont:D
\__kernel_primitive:NN \tojis                 \tex_tojis:D
\__kernel_primitive:NN \toucs                 \tex_toucs:D
\__kernel_primitive:NN \ucs                   \tex_ucs:D
\__kernel_primitive:NN \xkanjiskip            \tex_xkanjiskip:D
\__kernel_primitive:NN \xspcode               \tex_xspcode:D
\__kernel_primitive:NN \ybaselineshift        \tex_ybaselineshift:D
\__kernel_primitive:NN \yoko                  \tex_yoko:D
\__kernel_primitive:NN \vfi                   \tex_vfi:D
```

Primitives from upTEX.

```
  \__kernel_primitive:NN \disablecjktoken      \tex_disablecjktoken:D
  \__kernel_primitive:NN \enablecjktoken       \tex_enablecjktoken:D
  \__kernel_primitive:NN \forcecjktoken        \tex_forcecjktoken:D
  \__kernel_primitive:NN \kchar                \tex_kchar:D
  \__kernel_primitive:NN \kchardef             \tex_kchardef:D
  \__kernel_primitive:NN \uptexrevision        \tex_uptexrevision:D
  \__kernel_primitive:NN \uptexversion         \tex_uptexversion:D
```

Omega primitives provided by pTEX (listed separately mainly to allow understanding of their source).

```
  \__kernel_primitive:NN \odelcode             \tex_odelcode:D
  \__kernel_primitive:NN \odelimiter           \tex_odelimiter:D
  \__kernel_primitive:NN \omathaccent          \tex_omathaccent:D
  \__kernel_primitive:NN \omathchar            \tex_omathchar:D
  \__kernel_primitive:NN \omathchardef         \tex_omathchardef:D
  \__kernel_primitive:NN \omathcode            \tex_omathcode:D
  \__kernel_primitive:NN \oradical             \tex_oradical:D
```

Newer cross-engine primitives.

```
  \__kernel_primitive:NN \partokencontext      \tex_partokencontext:D
  \__kernel_primitive:NN \partokenname         \tex_partokenname:D
  \__kernel_primitive:NN \showstream           \tex_showstream:D
  \__kernel_primitive:NN \tracingstacklevels   \tex_tracingstacklevels:D
```

End of the “just the names” part of the source.

```
⟨/names | package⟩
⟨/names | tex⟩
⟨*package⟩
⟨*tex⟩
```

The job is done: close the group (using the primitive renamed!).

```
\tex_endgroup:D
```

LaTeX 2$_\varepsilon$ moves a few primitives, so these are sorted out. In newer versions of LaTeX 2$_\varepsilon$, expl3 is loaded rather early, so only some primitives are already renamed, so we need two tests here. At the beginning of the LaTeX 2$_\varepsilon$ format, the primitives `\end` and `\input` are renamed, and only later on the other ones.

```
\tex_ifdefined:D \@@end
  \tex_let:D \tex_end:D                 \@@end
  \tex_let:D \tex_input:D               \@@input
\tex_fi:D
```

If `\@@@@hyph` is defined, we are loading expl3 in a pre-2020/10/01 release of LaTeX 2$_\varepsilon$, so a few other primitives have to be tested as well.

```
\tex_ifdefined:D \@@hyph
  \tex_let:D \tex_everydisplay:D        \frozen@everydisplay
  \tex_let:D \tex_everymath:D           \frozen@everymath
  \tex_let:D \tex_hyphen:D              \@@hyph
  \tex_let:D \tex_italiccorrection:D    \@@italiccorr
  \tex_let:D \tex_underline:D           \@@underline
```

The `\shipout` primitive is particularly tricky as a number of packages want to hook in here. First, we see if a sufficiently-new kernel has saved a copy: if it has, just use that. Otherwise, we need to check each of the possible packages/classes that might move it: here, we are looking for those which do *not* delay action to the `\AtBeginDocument` hook. (We cannot use `\primitive` as that doesn't allow us to make a direct copy of the primitive *itself*.) As we know that LaTeX 2ε is in use, we use it's `\@tfor` loop here.

```
  \tex_ifdefined:D \@@shipout
    \tex_let:D \tex_shipout:D \@@shipout
  \tex_fi:D
  \tex_begingroup:D
    \tex_edef:D \l_tmpa_tl { \tex_string:D \shipout }
    \tex_edef:D \l_tmpb_tl { \tex_meaning:D \shipout }
    \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
    \tex_else:D
      \tex_expandafter:D \@tfor \tex_expandafter:D \@tempa \tex_string:D :=
        \CROP@shipout
        \dup@shipout
        \GPTorg@shipout
        \LL@shipout
        \mem@oldshipout
        \opem@shipout
        \pgfpages@originalshipout
        \pr@shipout
        \Shipout
        \verso@orig@shipout
        \do
          {
            \tex_edef:D \l_tmpb_tl
              { \tex_expandafter:D \tex_meaning:D \@tempa }
            \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
              \tex_global:D \tex_expandafter:D \tex_let:D
                \tex_expandafter:D \tex_shipout:D \@tempa
            \tex_fi:D
          }
    \tex_fi:D
  \tex_endgroup:D
```

Some tidying up is needed for `\(pdf)tracingfonts`. Newer LuaTeX has this simply as `\tracingfonts`, but that is overwritten by the LaTeX 2ε kernel. So any spurious definition has to be removed, then the real version saved either from the pdfTeX name or from LuaTeX. In the latter case, we leave `\@@tracingfonts` available: this might be useful and almost all LaTeX 2ε users will have expl3 loaded by fontspec. (We follow the usual kernel convention that `@@` is used for saved primitives.)

```
  \tex_let:D \tex_tracingfonts:D \tex_undefined:D
  \tex_ifdefined:D \pdftracingfonts
    \tex_let:D \tex_tracingfonts:D \pdftracingfonts
  \tex_else:D
    \tex_ifdefined:D \tex_directlua:D
      \tex_directlua:D { tex.enableprimitives("@@", {"tracingfonts"}) }
      \tex_let:D \tex_tracingfonts:D \@@tracingfonts
    \tex_fi:D
  \tex_fi:D
\tex_fi:D
```

Only pdfTeX and LuaTeX define `\pdfmapfile` and `\pdfmapline`: Tidy up the fact that some format-building processes leave a couple of questionable decisions about that!

```
\tex_ifnum:D 0
  \tex_ifdefined:D \tex_pdftexversion:D 1 \tex_fi:D
```

```
    \tex_ifdefined:D \tex_luatexversion:D 1 \tex_fi:D
      = 0 %
    \tex_let:D \tex_pdfmapfile:D \tex_undefined:D
    \tex_let:D \tex_pdfmapline:D \tex_undefined:D
  \tex_fi:D
```

A few packages do unfortunate things to date-related primitives.

```
  \tex_begingroup:D
    \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_time:D }
    \tex_edef:D \l_tmpb_tl { \tex_string:D \time }
    \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
    \tex_else:D
      \tex_global:D \tex_let:D \tex_time:D \tex_undefined:D
    \tex_fi:D
    \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_day:D }
    \tex_edef:D \l_tmpb_tl { \tex_string:D \day }
    \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
    \tex_else:D
      \tex_global:D \tex_let:D \tex_day:D \tex_undefined:D
    \tex_fi:D
    \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_month:D }
    \tex_edef:D \l_tmpb_tl { \tex_string:D \month }
    \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
    \tex_else:D
      \tex_global:D \tex_let:D \tex_month:D \tex_undefined:D
    \tex_fi:D
    \tex_edef:D \l_tmpa_tl { \tex_meaning:D \tex_year:D }
    \tex_edef:D \l_tmpb_tl { \tex_string:D \year }
    \tex_ifx:D \l_tmpa_tl \l_tmpb_tl
    \tex_else:D
      \tex_global:D \tex_let:D \tex_year:D \tex_undefined:D
    \tex_fi:D
  \tex_endgroup:D
```

cslatex moves a couple of primitives which we recover here; as there is no other marker, we can only work by looking for the names.

```
  \tex_ifdefined:D \orieveryjob
    \tex_let:D \tex_everyjob:D \orieveryjob
  \tex_fi:D
  \tex_ifdefined:D \oripdfoutput
    \tex_let:D \tex_pdfoutput:D \oripdfoutput
  \tex_fi:D
```

For ConTEXt, two tests are needed. Both Mark II and Mark IV move several primitives: these are all covered by the first test, again using `\end` as a marker. For Mark IV, a few more primitives are moved: they are implemented using some Lua code in the current ConTEXt.

```
  \tex_ifdefined:D \normalend
    \tex_let:D \tex_end:D         \normalend
    \tex_let:D \tex_everyjob:D    \normaleveryjob
    \tex_let:D \tex_input:D       \normalinput
    \tex_let:D \tex_language:D    \normallanguage
    \tex_let:D \tex_mathop:D      \normalmathop
    \tex_let:D \tex_month:D       \normalmonth
    \tex_let:D \tex_outer:D       \normalouter
```

```
  \tex_let:D \tex_over:D        \normalover
  \tex_let:D \tex_vcenter:D     \normalvcenter
  \tex_let:D \tex_unexpanded:D  \normalunexpanded
  \tex_let:D \tex_expanded:D    \normalexpanded
\tex_fi:D
\tex_ifdefined:D \normalitaliccorrection
  \tex_let:D \tex_hoffset:D          \normalhoffset
  \tex_let:D \tex_italiccorrection:D \normalitaliccorrection
  \tex_let:D \tex_voffset:D          \normalvoffset
  \tex_let:D \tex_showtokens:D       \normalshowtokens
  \tex_let:D \tex_bodydir:D          \spac_directions_normal_body_dir
  \tex_let:D \tex_pagedir:D          \spac_directions_normal_page_dir
\tex_fi:D
\tex_ifdefined:D \normalleft
  \tex_let:D \tex_left:D   \normalleft
  \tex_let:D \tex_middle:D \normalmiddle
  \tex_let:D \tex_right:D  \normalright
\tex_fi:D
```

⟨/tex⟩

In LuaTeX, we additionally emulate some primitives using Lua code.

⟨*lua⟩

`\tex_strcmp:D` Compare two strings, expanding to `0` if they are equal, `-1` if the first one is smaller and `1` if the second one is smaller. Here “smaller” refers to codepoint order which does not correspond to the user expected order for most non-ASCII strings.

```
local minus_tok = token_new(string.byte'-', 12)
local zero_tok = token_new(string.byte'0', 12)
local one_tok = token_new(string.byte'1', 12)
luacmd('tex_strcmp:D', function()
  local first = scan_string()
  local second = scan_string()
  if first < second then
    put_next(minus_tok, one_tok)
  else
    put_next(first == second and zero_tok or one_tok)
  end
end, 'global')
```

(*End of definition for* `\tex_strcmp:D`.)

`\tex_Ucharcat:D` Creating arbitrary chars using `tex.cprint`. The alternative approach using `token.new(...)` is about 10% slower but needed to create arbitrary space tokens.

```
local sprint = tex.sprint
local cprint = tex.cprint
luacmd('tex_Ucharcat:D', function()
  local charcode = scan_int()
  local catcode = scan_int()
  if catcode == 10 then
    sprint(token_new(charcode, 10))
  else
    cprint(catcode, utf8_char(charcode))
  end
end, 'global')
```

(*End of definition for* `\tex_Ucharcat:D`*.*)

`\tex_filesize:D` Wrap the function from `ltxutils`.

```
luacmd('tex_filesize:D', function()
  local size = filesize(scan_string())
  if size then write(size) end
end, 'global')
```

(*End of definition for* `\tex_filesize:D`*.*)

`\tex_mdfivesum:D` There are two cases: Either hash a file or a string. Both are already implemented in l3luatex or built-in.

```
luacmd('tex_mdfivesum:D', function()
  local hash
  if scan_keyword"file" then
    hash = filemd5sum(scan_string())
  else
    hash = md5_HEX(scan_string())
  end
  if hash then write(hash) end
end, 'global')
```

(*End of definition for* `\tex_mdfivesum:D`*.*)

`\tex_filemoddate:D` A primitive for getting the modification date of a file.

```
luacmd('tex_filemoddate:D', function()
  local date = filemoddate(scan_string())
  if date then write(date) end
end, 'global')
```

(*End of definition for* `\tex_filemoddate:D`*.*)

`\tex_filedump:D` An emulated primitive for getting a hexdump from a (partial) file. The length has a default of `0`. This is consistent with pdfTEX, but it effectively makes the primitive useless without an explicit `length`. Therefore we allow the keyword `whole` to be used instead of a length, indicating that the whole remaining file should be read.

```
luacmd('tex_filedump:D', function()
  local offset = scan_keyword'offset' and scan_int() or nil
  local length = scan_keyword'length' and scan_int()
          or not scan_keyword'whole' and 0 or nil
  local data = filedump(scan_string(), offset, length)
  if data then write(data) end
end, 'global')
```

(*End of definition for* `\tex_filedump:D`*.*)

```
⟨/lua⟩
⟨/package⟩
```

Chapter 41

l3kernel-functions: kernel-reserved functions

41.1 Internal l3debug kernel functions

These function are only created if debugging is enabled, hence they are actually defined in l3debug.

`\__kernel_chk_var_local:N`
`\__kernel_chk_var_global:N`

`\__kernel_chk_var_local:N` ⟨*var*⟩
`\__kernel_chk_var_global:N` ⟨*var*⟩

Applies `\__kernel_chk_var_exist:N` ⟨*var*⟩ as well as `\__kernel_chk_var_scope:NN` ⟨*scope*⟩ ⟨*var*⟩, where ⟨*scope*⟩ is `l` or `g`.

`\__kernel_chk_var_scope:NN`

`\__kernel_chk_var_scope:NN` ⟨*scope*⟩ ⟨*var*⟩

Checks the ⟨*var*⟩ has the correct ⟨*scope*⟩, and if not raises a kernel-level error. The ⟨*scope*⟩ is a single letter `l`, `g`, `c` denoting local variables, global variables, or constants. More precisely, if the variable name starts with a letter and an underscore (normal expl3 convention) the function checks that this single letter matches the ⟨*scope*⟩. Otherwise the function cannot know the scope ⟨*var*⟩ the first time: instead, it defines `\__debug_chk_/`⟨*var name*⟩ to store that information for the next call. Thus, if a given ⟨*var*⟩ is subject to assignments of different scopes a kernel error will result.

`\__kernel_chk_cs_exist:N`
`\__kernel_chk_cs_exist:c`
`\__kernel_chk_var_exist:N`

`\__kernel_chk_cs_exist:N` ⟨*cs*⟩
`\__kernel_chk_var_exist:N` ⟨*var*⟩

Checks that their argument is defined according to the criteria for `\cs_if_exist_p:N`, and if not raises a kernel-level error. Error messages are different.

`\__kernel_chk_flag_exist:NN` ⋆

`\__kernel_chk_flag_exist:NN` ⟨*function*⟩ ⟨*flag*⟩

Checks that the ⟨*flag*⟩ is defined according to the criterion for `\flag_if_exist_p:N`, and if not raises a kernel-level error and calls the function with the argument `\l_tmpa_flag` to proceed somehow without producing too many errors.

__kernel_debug_log:e __kernel_debug_log:e {⟨*message text*⟩}

If the log-functions option is active, this function writes the ⟨*message text*⟩ to the log file using \iow_log:e. Otherwise, the ⟨*message text*⟩ is ignored using \use_none:n.

41.2 Internal kernel functions

__kernel_chk_defined:NT __kernel_chk_defined:NT ⟨*variable*⟩ {⟨*true code*⟩}

If ⟨*variable*⟩ is not defined (according to \cs_if_exist:NTF), this triggers an error, otherwise the ⟨*true code*⟩ is run.

__kernel_chk_expr:nNnN __kernel_chk_expr:nNnN {⟨*expr*⟩} ⟨*eval*⟩ {⟨*convert*⟩} ⟨*caller*⟩

This function is only created if debugging is enabled. By default it is equivalent to \use_i:nnnn. When expression checking is enabled, it leaves in the input stream the result of \tex_the:D ⟨*eval*⟩ ⟨*expr*⟩ \tex_relax:D after checking that no token was left over. If any token was not taken as part of the expression, there is an error message displaying the result of the evaluation as well as the ⟨*caller*⟩. For instance ⟨*eval*⟩ can be __int_eval:w and ⟨*caller*⟩ can be \int_eval:n or \int_set:Nn. The argument ⟨*convert*⟩ is empty except for mu expressions where it is \tex_mutoglue:D, used for internal purposes.

__kernel_chk_tl_type:NnnT __kernel_chk_tl_type:NnnT ⟨*control sequence*⟩ {⟨*specific type*⟩} {⟨*reconstruction*⟩} {⟨*true code*⟩}

Helper to test that the ⟨*control sequence*⟩ is a variable of the given ⟨*specific type*⟩ of token list. Produces suitable error messages if the ⟨*control sequence*⟩ does not exist, or if it is not a token list variable at all, or if the ⟨*control sequence*⟩ differs from the result of e-expanding ⟨*reconstruction*⟩. If all of these tests succeed then the ⟨*true code*⟩ is run.

__kernel_codepoint_to_bytes:n ⋆ __kernel_codepoint_to_bytes:n {⟨*codepoint*⟩}

Converts the ⟨*codepoint*⟩ to UTF-8 bytes. The expansion of this function comprises four brace groups, each of which will contain a hexadecimal value: the appropriate byte. As UTF-8 is a variable-length, one or more of the groups may be empty: the bytes read in the logical order, such that a two-byte codepoint will have groups #1 and #2 filled and #3 and #4 empty.

__kernel_cs_parm_from_arg_count:nnF __kernel_cs_parm_from_arg_count:nnF {⟨*follow-on*⟩} {⟨*args*⟩} {⟨*false code*⟩}

Evaluates the number of ⟨*args*⟩ and leaves the ⟨*follow-on*⟩ code followed by a brace group containing the required number of primitive parameter markers (#1, *etc.*). If the number of ⟨*args*⟩ is outside the range $[0, 9]$, the ⟨*false code*⟩ is inserted *instead* of the ⟨*follow-on*⟩.

`\__kernel_dependency_version_check:Nn` `\__kernel_dependency_version_check:Nn {⟨\date⟩} {⟨file⟩}`
`\__kernel_dependency_version_check:nn` `\__kernel_dependency_version_check:nn {⟨date⟩} {⟨file⟩}`

Checks if the loaded version of the expl3 kernel is at least ⟨*date*⟩, required by ⟨*file*⟩. If the kernel date is older than ⟨*date*⟩, the loading of ⟨*file*⟩ is aborted and an error is raised.

`\__kernel_deprecation_code:nn` `\__kernel_deprecation_code:nn {⟨error code⟩} {⟨working code⟩}`

Stores both an ⟨*error*⟩ and ⟨*working*⟩ definition for given material such that they can be exchanged by `\debug_on:n` and `\debug_off:n`.

`\__kernel_exp_not:w` ⋆ `\__kernel_exp_not:w ⟨expandable tokens⟩ {⟨content⟩}`

Carries out expansion on the ⟨*expandable tokens*⟩ before preventing further expansion of the ⟨*content*⟩ as for `\exp_not:n`. Typically, the ⟨*expandable tokens*⟩ will alter the nature of the ⟨*content*⟩, *i.e.* allow it to be generated in some way.

`\l__kernel_expl_bool` A boolean which records the current code syntax status: `true` if currently inside a code environment. This variable should only be set by `\ExplSyntaxOn`/`\ExplSyntaxOff`.

(*End of definition for* `\l__kernel_expl_bool`.)

`\c__kernel_expl_date_tl` A token list containing the release date of the l3kernel preloaded in LaTeX 2ε used to check if dependencies match.

(*End of definition for* `\c__kernel_expl_date_tl`.)

`\__kernel_file_missing:n` `\__kernel_file_missing:n {⟨name⟩}`

Expands the ⟨*name*⟩ as per `\__kernel_file_name_sanitize:n` then produces an error message indicating that this file was not found.

`\__kernel_file_name_sanitize:n` ⋆ `\__kernel_file_name_sanitize:n {⟨name⟩}`

Updated: 2021-04-17

Expands the file name using a `\csname`-based approach, and relies on active characters (for example from UTF-8 characters) being properly set up to expand to a expansion-safe version using `\ifcsname`. This is less conservative than the token-by-token approach used before, but it is much faster.

`\__kernel_file_input_push:n` `\__kernel_file_input_push:n {⟨name⟩}`
`\__kernel_file_input_pop:` `\__kernel_file_input_pop:`

Used to push and pop data from the internal file stack: needed only in package mode, where interfacing with the LaTeX 2ε kernel is necessary.

`\__kernel_int_add:nnn` ⋆ `\__kernel_int_add:nnn {⟨integer₁⟩} {⟨integer₂⟩} {⟨integer₃⟩}`

Expands to the result of adding the three ⟨*integers*⟩ (which must be suitable input for `\int_eval:w`), avoiding intermediate overflow. Overflow occurs only if the overall result is outside $[-2^{31}+1, 2^{31}-1]$. The ⟨*integers*⟩ may be of the form `\int_eval:w` … `\scan_stop:` but may be evaluated more than once.

__kernel_intarray_gset:Nnn __kernel_intarray_gset:Nnn ⟨*intarray var*⟩ {⟨*index*⟩} {⟨*value*⟩}

New: 2018-03-31

Faster version of \intarray_gset:Nnn. Stores the ⟨*value*⟩ into the ⟨*integer array variable*⟩ at the ⟨*position*⟩. The ⟨*index*⟩ and ⟨*value*⟩ must be suitable for a direct assignment to a TeX count register, for instance expanding to an integer denotation or obtained through the primitive \numexpr (which may be un-terminated). No bound checking is performed: the caller is responsible for ensuring that the ⟨*position*⟩ is between 1 and the \intarray_count:N, and the ⟨*value*⟩'s absolute value is at most $2^{30}-1$. Assignments are always global.

__kernel_intarray_item:Nn ⋆ __kernel_intarray_item:Nn ⟨*intarray var*⟩ {⟨*index*⟩}

New: 2018-03-31

Faster version of \intarray_item:Nn. Expands to the integer entry stored at the ⟨*index*⟩ in the ⟨*integer array variable*⟩. The ⟨*index*⟩ must be suitable for a direct assignment to a TeX count register and must be between 1 and the \intarray_count:N, lest a low-level TeX error occur.

__kernel_intarray_range_to_clist:Nnn ☆ __kernel_intarray_range_to_clist:Nnn ⟨*intarray var*⟩ {⟨*start index*⟩} {⟨*end index*⟩}

New: 2020-07-12

Converts to integer denotations separated by commas the entries of the ⟨*intarray*⟩ from positions ⟨*start index*⟩ to ⟨*end index*⟩ included. The ⟨*start index*⟩ and ⟨*end index*⟩ must be suitable for a direct assignment to a TeX count register, must be between 1 and the \intarray_count:N, and be suitably ordered. All tokens have category code other.

__kernel_intarray_gset_range_from_clist:Nnn __kernel_intarray_gset_range_from_clist:Nnn ⟨*intarray var*⟩ {⟨*start index*⟩} {⟨*integer clist*⟩}

New: 2020-07-12

Stores the entries of the ⟨*clist*⟩ as entries of the ⟨*intarray var*⟩ starting from the ⟨*start index*⟩, upwards. This is done without any bound checking. The ⟨*start index*⟩ and all entries of the ⟨*integer comma list*⟩ (which do not undergo space trimming and brace stripping as in normal clist mappings) must be suitable for a direct assignment to a TeX count register. An empty entry may stop the loop.

__kernel_ior_open:Nn
__kernel_ior_open:No

__kernel_ior_open:Nn ⟨*stream*⟩ {⟨*file name*⟩}

This function has identical syntax to the public version. However, is does not take precautions against active characters in the ⟨*file name*⟩, and it does not attempt to add a ⟨*path*⟩ to the ⟨*file name*⟩: it is therefore intended to be used by higher-level functions which have already fully expanded the ⟨*file name*⟩ and which need to perform multiple open or close operations. See for example the implementation of \ior_shell_open:Nn.

__kernel_iow_open:Nn
__kernel_iow_open:No

__kernel_iow_open:Nn ⟨*stream*⟩ {⟨*file name*⟩}

This function has identical syntax to the public version. However, is does not take precautions against active characters in the ⟨*file name*⟩, and it does not attempt to add a ⟨*path*⟩ to the ⟨*file name*⟩: it is therefore intended to be used by higher-level functions which have already fully expanded the ⟨*file name*⟩ and which need to perform multiple open or close operations. See for example the implementation of \iow_shell_open:Nn.

__kernel_iow_with:Nnn __kernel_iow_with:Nnn ⟨*integer*⟩ {⟨*value*⟩} {⟨*code*⟩}

If the ⟨*integer*⟩ is equal to the ⟨*value*⟩ then this function simply runs the ⟨*code*⟩. Otherwise it saves the current value of the ⟨*integer*⟩, sets it to the ⟨*value*⟩, runs the ⟨*code*⟩, and restores the ⟨*integer*⟩ to its former value. This is used to ensure that the \newlinechar is 10 when writing to a stream, which lets \iow_newline: work, and that \errorcontextlines is -1 when displaying a message.

__kernel_kern:n __kernel_kern:n {⟨*length*⟩}

Inserts a kern of the specified ⟨*length*⟩, a dimension expression.

(*End of definition for* __kernel_kern:n.)

__kernel_msg_show_eval:Nn __kernel_msg_show_eval:Nn ⟨*function*⟩ {⟨*expression*⟩}
__kernel_msg_log_eval:Nn

Shows or logs the ⟨*expression*⟩ (turned into a string), an equal sign, and the result of applying the ⟨*function*⟩ to the {⟨*expression*⟩} (with f-expansion). For instance, if the ⟨*function*⟩ is \int_eval:n and the ⟨*expression*⟩ is 1+2 then this logs > 1+2=3.

__kernel_pdf_object_id:n ⋆ __kernel_pdf_object_id:n {⟨*object*⟩}
__kernel_pdf_object_id_indexed:nn ⋆ __kernel_pdf_object_id_indexed:nn {⟨*class*⟩} {⟨*number*⟩}

Expands to the ID of ⟨*object*⟩ (or object of ⟨*number*⟩ within the ⟨*class*⟩), in for example page resource allocation. Depending on the backend, the result may be the same as \pdf_object_id:n/\pdf_object_id_indexed:nn.

\g__kernel_prg_map_int This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions \⟨*type*⟩_map_1:w, \⟨*type*⟩_map_2:w, *etc.*, labelled by \g__kernel_prg_map_int hold functions to be mapped over various list datatypes in inline and variable mappings.

(*End of definition for* \g__kernel_prg_map_int.)

`\__kernel_quark_new_test:N` `\__kernel_quark_new_test:N \⟨name⟩:⟨arg spec⟩`

Defines a quark-test function `\⟨name⟩:⟨arg spec⟩` which tests if its argument is `\q__⟨namespace⟩_recursion_tail`, then acts accordingly, as described below for each possible ⟨arg spec⟩.

The ⟨namespace⟩ is determined as the first (nonempty) `_`-delimited word in ⟨name⟩ and is used internally in the definition of auxiliaries. The function `\__kernel_quark_new_test:N` does *not* define the `\q__⟨namespace⟩_recursion_tail` and `\q__⟨namespace⟩_recursion_stop` quarks. They should be manually defined with `\quark_new:N`.

There are 6 different types of quark-test functions. Which one is defined depends on the ⟨arg spec⟩, which *must* be one of the options listed now. Four of them are modeled after `\quark_if_recursion_tail:(N|n)` and `\quark_if_recursion_tail_do:(N|n)n`.

`n` defines `\⟨name⟩:n` such that it checks if `#1` contains only `\q__⟨namespace⟩_recursion_tail`, and if so consumes all tokens up to `\q__⟨namespace⟩_recursion_stop` (*c.f.* `\quark_if_recursion_tail_stop:n`).

`nn` defines `\⟨name⟩:nn` such that it checks if `#1` contains only `\q__⟨namespace⟩_recursion_tail`, and if so consumes all tokens up to `\q__⟨namespace⟩_recursion_stop`, then executes the code `#2` after that (*c.f.* `\quark_if_recursion_tail_stop_do:nn`).

`N` defines `\⟨name⟩:N` such that it checks if `#1` is `\q__⟨namespace⟩_recursion_tail`, and if so consumes all tokens up to `\q__⟨namespace⟩_recursion_stop` (*c.f.* `\quark_if_recursion_tail_stop:N`).

`Nn` defines `\⟨name⟩:Nn` such that it checks if `#1` is `\q__⟨namespace⟩_recursion_tail`, and if so consumes all tokens up to `\q__⟨namespace⟩_recursion_stop`, then executes the code `#2` after that (*c.f.* `\quark_if_recursion_tail_stop_do:Nn`).

The last two are modeled after `\quark_if_recursion_tail_break:(n|N)N`, and in those cases the quark `\q__⟨namespace⟩_recursion_stop` is not used (and thus needs not be defined).

`nN` defines `\⟨name⟩:nN` such that it checks if `#1` contains only `\q__⟨namespace⟩_recursion_tail`, and if so uses the `\⟨type⟩_map_break:` function `#2`.

`NN` defines `\⟨name⟩:NN` such that it checks if `#1` is `\q__⟨namespace⟩_recursion_tail`, and if so uses the `\⟨type⟩_map_break:` function `#2`.

Any other signature, as well as a function without signature are errors, and in such case the definition is aborted.

`\__kernel_quark_new_conditional:Nn` `\__kernel_quark_new_conditional:Nn \__⟨namespace⟩_quark_if_⟨name⟩:⟨arg spec⟩ {⟨conditions⟩}`

Defines a collection of quark conditionals that test if their argument is the quark `\q__⟨namespace⟩_⟨name⟩` and perform suitable actions. The ⟨*conditions*⟩ are a comma-separated list of one or more of `p`, `T`, `F`, and `TF`, and one conditional is defined for each ⟨*condition*⟩ in the list, as described for `\prg_new_conditional:Npnn`. The conditionals are defined using `\prg_new_conditional:Npnn`, so that their name is obtained by adding `p`, `T`, `F`, or `TF` to the base name `\__⟨namespace⟩_quark_if_⟨name⟩:⟨arg spec⟩`.

The first argument of `\__kernel_quark_new_conditional:Nn` must contain `_quark_if_` and `:`, as these markers are used to determine the ⟨*name*⟩ of the quark `\q__⟨namespace⟩_⟨name⟩` to be tested. This quark should be manually defined with `\quark_new:N`, as `\__kernel_quark_new_conditional:Nn` does *not* define it.

The function `\__kernel_quark_new_conditional:Nn` can define 2 different types of quark conditionals. Which one is defined depends on the ⟨*arg spec*⟩, which *must* be one of the following options, modeled after `\quark_if_nil:(N|n)(TF)`.

`n` defines `\__⟨namespace⟩_quark_if_⟨name⟩:n(TF)` such that it checks if `#1` contains only `\q__⟨namespace⟩_⟨name⟩`, and executes the proper conditional branch.

`N` defines `\__⟨namespace⟩_quark_if_⟨name⟩:N(TF)` such that it checks if `#1` is `\q__⟨namespace⟩_⟨name⟩`, and executes the proper conditional branch.

Any other signature, as well as a function without signature are errors, and in such case the definition is aborted.

`\__kernel_sys_everyjob:` `\__kernel_sys_everyjob:`

Inserts the internal token list required at the start of every run (job).

`\c__kernel_randint_max_int` Maximal allowed argument to `\__kernel_randint:n`. Equal to $2^{17} - 1$.

(*End of definition for* `\c__kernel_randint_max_int`*.*)

`\__kernel_randint:n` `\__kernel_randint:n {⟨max⟩}`

Used in an integer expression this gives a pseudo-random number between 1 and ⟨*max*⟩ included. One must have ⟨*max*⟩ $\leq 2^{17} - 1$. The ⟨*max*⟩ must be suitable for `\int_value:w` (and any `\int_eval:w` must be terminated by `\scan_stop:` or equivalent).

`\__kernel_randint:nn` `\__kernel_randint:nn {⟨min⟩} {⟨max⟩}`

Used in an integer expression this gives a pseudo-random number between ⟨*min*⟩ and ⟨*max*⟩ included. The ⟨*min*⟩ and ⟨*max*⟩ must be suitable for `\int_value:w` (and any `\int_eval:w` must be terminated by `\scan_stop:` or equivalent). For small ranges $R = \langle max\rangle - \langle min\rangle + 1 \leq 2^{17} - 1$, $\langle min\rangle - 1 +$ `\__kernel_randint:n{`R`}` is faster.

`\__kernel_register_show:N`
`\__kernel_register_show:c`
`\__kernel_register_show:N ⟨register⟩`

Used to show the contents of a TeX register at the terminal, formatted such that internal parts of the mechanism are not visible.

`\__kernel_register_log:N`
`\__kernel_register_log:c`
`\__kernel_register_log:N ⟨register⟩`

Used to write the contents of a TeX register to the log file in a form similar to `\__kernel_register_show:N`.

`\__kernel_str_to_other:n` ⋆ `\__kernel_str_to_other:n {⟨token list⟩}`

Converts the ⟨*token list*⟩ to a ⟨*other string*⟩, where spaces have category code "other". This function can be `f`-expanded without fear of losing a leading space, since spaces do not have category code 10 in its result. It takes a time quadratic in the character count of the string.

`\__kernel_str_to_other_fast:n` ☆ `\__kernel_str_to_other_fast:n {⟨token list⟩}`

Same behaviour `\__kernel_str_to_other:n` but only restricted-expandable. It takes a time linear in the character count of the string.

`\__kernel_tl_to_str:w` ⋆ `\__kernel_tl_to_str:w ⟨expandable tokens⟩ {⟨tokens⟩}`

Carries out expansion on the ⟨*expandable tokens*⟩ before conversion of the ⟨*tokens*⟩ to a string as describe for `\tl_to_str:n`. Typically, the ⟨*expandable tokens*⟩ will alter the nature of the ⟨*tokens*⟩, *i.e.* allow it to be generated in some way. This function requires only a single expansion.

`\__kernel_tl_set:Nx`
`\__kernel_tl_gset:Nx`
`\__kernel_tl_set:Nx ⟨tl var⟩ {⟨tokens⟩}`

Fully expands ⟨*tokens*⟩ and assigns the result to ⟨*tl var*⟩. ⟨*tokens*⟩ must be given in braces and there must be no token between ⟨*tl var*⟩ and ⟨*tokens*⟩.

`\__kernel_codepoint_data:nn` ⋆ `\__kernel_codepoint_data:nn {⟨type⟩} {⟨codepoint⟩}`

Expands to the appropriate value for the ⟨*type*⟩ of data requested for a ⟨*codepoint*⟩. The current list of ⟨*types*⟩ and results are

`lowercase` The *single* codepoint specified by `UnicodeData.txt` for lowercase mapping of the codepoint: will be equal to the input ⟨*codepoint*⟩ if there is no mapping specified in `UnicodeData.txt`

`uppercase` The *single* codepoint specified by `UnicodeData.txt` for uppercase mapping of the codepoint: will be equal to the input ⟨*codepoint*⟩ if there is no mapping specified in `UnicodeData.txt`

`\__kernel_codepoint_case:nn` ⋆ `\__kernel_codepoint_case:nn {⟨mapping⟩} {⟨codepoint⟩}`

Expands to a list of three balanced text, of which at least the first will contain a codepoint. This list of up to three codepoints specifies the full case mapping for the input ⟨*codepoint*⟩. The ⟨*mapping*⟩ should be one of

- `casefold`
- `lowercase`
- `titlecase`
- `uppercase`

41.3 Kernel backend functions

These functions are required to pass information to the backend. The nature of these means that they are defined only when the relevant backend is in use.

`\__kernel_backend_literal:n`
`\__kernel_backend_literal:(e|e)`

`\__kernel_backend_literal:n {⟨content⟩}`

Adds the ⟨*content*⟩ literally to the current vertical list as a whatsit. The nature of the ⟨*content*⟩ will depend on the backend in use.

`\__kernel_backend_literal_postscript:n`
`\__kernel_backend_literal_postscript:e`

`\__kernel_backend_literal_postscript:n {⟨PostScript⟩}`

Adds the ⟨*PostScript*⟩ literally to the current vertical list as a whatsit. No positioning is applied.

`\__kernel_backend_literal_pdf:n`
`\__kernel_backend_literal_pdf:e`

`\__kernel_backend_literal_pdf:n {⟨PDF instructions⟩}`

Adds the ⟨*PDF instructions*⟩ literally to the current vertical list as a whatsit. No positioning is applied.

`\__kernel_backend_literal_svg:n`
`\__kernel_backend_literal_svg:e`

`\__kernel_backend_literal_svg:n {⟨SVG instructions⟩}`

Adds the ⟨*SVG instructions*⟩ literally to the current vertical list as a whatsit. No positioning is applied.

`\__kernel_backend_postscript:n`
`\__kernel_backend_postscript:e`

`\__kernel_backend_postscript:n {⟨PostScript⟩}`

Adds the ⟨*PostScript*⟩ to the current vertical list as a whatsit. The PostScript reference point is adjusted to match the current position. The PostScript is inserted inside a `SDict begin`/`end` pair.

`\__kernel_backend_align_begin:`
`\__kernel_backend_align_end:`

```
\__kernel_backend_align_begin:
⟨PostScript literals⟩
\__kernel_backend_align_end:
```

Arranges to align the PostScript and DVI current positions and scales.

`\__kernel_backend_scope_begin:`
`\__kernel_backend_scope_end:`

```
\__kernel_backend_scope_begin:
⟨content⟩
\__kernel_backend_scope_end:
```

Creates a scope for instructions at the backend level.

`\__kernel_backend_matrix:n`
`\__kernel_backend_matrix:e`

`\__kernel_backend_matrix:n {⟨matrix⟩}`

Applies the ⟨*matrix*⟩ to the current transformation matrix.

`\g__kernel_backend_header_bool`

Specifies whether to write headers for the backend.

`\l__kernel_color_stack_int` The color stack used in pdfTEX and LuaTEX for the main color.

Chapter 42

l3basics implementation

```
⟨*package⟩
```

42.1 Renaming some TeX primitives (again)

Having given all the TeX primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but we do a few now, just to get started.[8]

\if_true:
\if_false:
\or:
\else:
\fi:
\reverse_if:N
\if:w
\if_charcode:w
\if_catcode:w
\if_meaning:w

Then some conditionals.

```
\tex_global:D \tex_let:D \if_true:           \tex_iftrue:D
\tex_global:D \tex_let:D \if_false:          \tex_iffalse:D
\tex_global:D \tex_let:D \or:                \tex_or:D
\tex_global:D \tex_let:D \else:              \tex_else:D
\tex_global:D \tex_let:D \fi:                \tex_fi:D
\tex_global:D \tex_let:D \reverse_if:N       \tex_unless:D
\tex_global:D \tex_let:D \if:w               \tex_if:D
\tex_global:D \tex_let:D \if_charcode:w      \tex_if:D
\tex_global:D \tex_let:D \if_catcode:w       \tex_ifcat:D
\tex_global:D \tex_let:D \if_meaning:w       \tex_ifx:D
\tex_global:D \tex_let:D \if_bool:N          \tex_ifodd:D
```

(*End of definition for* `\if_true:` *and others. These functions are documented on page 29.*)

\if_mode_math:
\if_mode_horizontal:
\if_mode_vertical:
\if_mode_inner:

TeX lets us detect some if its modes.

```
\tex_global:D \tex_let:D \if_mode_math:       \tex_ifmmode:D
\tex_global:D \tex_let:D \if_mode_horizontal: \tex_ifhmode:D
\tex_global:D \tex_let:D \if_mode_vertical:   \tex_ifvmode:D
\tex_global:D \tex_let:D \if_mode_inner:      \tex_ifinner:D
```

(*End of definition for* `\if_mode_math:` *and others. These functions are documented on page 30.*)

\if_cs_exist:N
\if_cs_exist:w
\cs:w
\cs_end:

Building csnames and testing if control sequences exist.

```
\tex_global:D \tex_let:D \if_cs_exist:N      \tex_ifdefined:D
\tex_global:D \tex_let:D \if_cs_exist:w      \tex_ifcsname:D
\tex_global:D \tex_let:D \cs:w               \tex_csname:D
\tex_global:D \tex_let:D \cs_end:            \tex_endcsname:D
```

[8]This renaming gets expensive in terms of csname usage, an alternative scheme would be to just use the `\tex_...:D` name in the cases where no good alternative exists.

(*End of definition for* `\if_cs_exist:N` *and others. These functions are documented on page 30.*)

`\exp_after:wN`
`\exp_not:N`
`\exp_not:n`

The five `\exp_` functions are used in the l3expan module where they are described.

```
\tex_global:D \tex_let:D \exp_after:wN      \tex_expandafter:D
\tex_global:D \tex_let:D \exp_not:N         \tex_noexpand:D
\tex_global:D \tex_let:D \exp_not:n         \tex_unexpanded:D
\tex_global:D \tex_let:D \exp:w             \tex_romannumeral:D
\tex_global:D \tex_chardef:D \exp_end:  = 0 ~
```

(*End of definition for* `\exp_after:wN`*,* `\exp_not:N`*, and* `\exp_not:n`*. These functions are documented on page 40.*)

`\token_to_meaning:N`
`\cs_meaning:N`

Examining a control sequence or token.

```
\tex_global:D \tex_let:D \token_to_meaning:N \tex_meaning:D
\tex_global:D \tex_let:D \cs_meaning:N       \tex_meaning:D
```

(*End of definition for* `\token_to_meaning:N` *and* `\cs_meaning:N`*. These functions are documented on page 200.*)

`\tl_to_str:n`
`\token_to_str:N`
`\__kernel_tl_to_str:w`

Making strings.

```
\tex_global:D \tex_let:D \tl_to_str:n          \tex_detokenize:D
\tex_global:D \tex_let:D \token_to_str:N       \tex_string:D
\tex_global:D \tex_let:D \__kernel_tl_to_str:w \tex_detokenize:D
```

(*End of definition for* `\tl_to_str:n`*,* `\token_to_str:N`*, and* `\__kernel_tl_to_str:w`*. These functions are documented on page 116.*)

`\scan_stop:`
`\group_begin:`
`\group_end:`

The next three are basic functions for which there also exist versions that are safe inside alignments. These safe versions are defined in the l3prg module.

```
\tex_global:D \tex_let:D \scan_stop:        \tex_relax:D
\tex_global:D \tex_let:D \group_begin:      \tex_begingroup:D
\tex_global:D \tex_let:D \group_end:        \tex_endgroup:D
```

(*End of definition for* `\scan_stop:`*,* `\group_begin:`*, and* `\group_end:`*. These functions are documented on page 14.*)

```
⟨@@=int⟩
```

`\if_int_compare:w`
`\__int_to_roman:w`

For integers.

```
\tex_global:D \tex_let:D \if_int_compare:w   \tex_ifnum:D
\tex_global:D \tex_let:D \__int_to_roman:w    \tex_romannumeral:D
```

(*End of definition for* `\if_int_compare:w` *and* `\__int_to_roman:w`*. This function is documented on page 178.*)

`\group_insert_after:N`

Adding material after the end of a group.

```
\tex_global:D \tex_let:D \group_insert_after:N \tex_aftergroup:D
```

(*End of definition for* `\group_insert_after:N`*. This function is documented on page 15.*)

`\exp_args:Nc`
`\exp_args:cc`

Discussed in l3expan, but needed much earlier.

```
\tex_long:D \tex_gdef:D \exp_args:Nc #1#2
  { \exp_after:wN #1 \cs:w #2 \cs_end: }
\tex_long:D \tex_gdef:D \exp_args:cc #1#2
  { \cs:w #1 \exp_after:wN \cs_end: \cs:w #2 \cs_end: }
```

(*End of definition for* `\exp_args:Nc` *and* `\exp_args:cc`*. These functions are documented on page 37.*)

\token_to_meaning:c
\token_to_str:c
\cs_meaning:c

A small number of variants defined by hand. Some of the necessary functions (\use_-i:nn, \use_ii:nn, and \exp_args:NNc) are not defined at that point yet, but will be defined before those variants are used. The \cs_meaning:c command must check for an undefined control sequence to avoid defining it mistakenly.

```
\tex_gdef:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
\tex_long:D \tex_gdef:D \cs_meaning:c #1
  {
    \if_cs_exist:w #1 \cs_end:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
    { \exp_args:Nc \cs_meaning:N {#1} }
    { \tl_to_str:n {undefined} }
  }
\tex_global:D \tex_let:D \token_to_meaning:c = \cs_meaning:c
```

(*End of definition for* \token_to_meaning:N. *This function is documented on page 200.*)

42.2 Defining some constants

\c_zero_int

We need the constant \c_zero_int which is used by some functions in current module. The rest are defined in the l3int module – at least for the ones that can be defined with \tex_chardef:D or \tex_mathchardef:D. For other constants the l3int module is required but it can't be used until the allocation has been set up properly!

```
\tex_global:D \tex_chardef:D \c_zero_int    = 0 ~
```

(*End of definition for* \c_zero_int. *This variable is documented on page 177.*)

\c_max_register_int

This is here as this particular integer is needed in modules loaded before l3int, and is documented in l3int. LuaTeX and those which contain parts of the Omega extensions have more registers available than ε-TeX.

```
\tex_ifdefined:D \tex_luatexversion:D
  \tex_global:D \tex_chardef:D \c_max_register_int = 65 535 ~
\tex_else:D
  \tex_ifdefined:D \tex_omathchardef:D
    \tex_global:D \tex_omathchardef:D \c_max_register_int = 65535 ~
  \tex_else:D
    \tex_global:D \tex_mathchardef:D \c_max_register_int = 32767 ~
  \tex_fi:D
\tex_fi:D
```

(*End of definition for* \c_max_register_int. *This variable is documented on page 177.*)

42.3 Defining functions

We start by providing functions for the typical definition functions. First the global ones.

\cs_gset_nopar:Npn
\cs_gset_nopar:Npe
\cs_gset_nopar:Npx
\cs_gset:Npn
\cs_gset:Npe
\cs_gset:Npx
\cs_gset_protected_nopar:Npn
\cs_gset_protected_nopar:Npe
\cs_gset_protected_nopar:Npx
\cs_gset_protected:Npn
\cs_gset_protected:Npe
\cs_gset_protected:Npx

All assignment functions in LaTeX3 should be naturally protected; after all, the TeX primitives for assignments are and it can be a cause of problems if others aren't.

```
\tex_global:D \tex_let:D \cs_gset_nopar:Npn           \tex_gdef:D
\tex_global:D \tex_let:D \cs_gset_nopar:Npe           \tex_xdef:D
```

```
\tex_global:D \tex_let:D \cs_gset_nopar:Npx          \tex_xdef:D
\tex_protected:D \tex_long:D \tex_gdef:D \cs_gset:Npn
  { \tex_long:D \tex_gdef:D }
\tex_protected:D \tex_long:D \tex_gdef:D \cs_gset:Npe
  { \tex_long:D \tex_xdef:D }
\tex_global:D \tex_let:D \cs_gset:Npx \cs_gset:Npe
\tex_protected:D \tex_long:D \tex_gdef:D \cs_gset_protected_nopar:Npn
  { \tex_protected:D \tex_gdef:D }
\tex_protected:D \tex_long:D \tex_gdef:D \cs_gset_protected_nopar:Npe
  { \tex_protected:D \tex_xdef:D }
\tex_global:D \tex_let:D \cs_gset_protected_nopar:Npx \cs_gset_protected_nopar:Npe
\tex_protected:D \tex_long:D \tex_gdef:D \cs_gset_protected:Npn
  { \tex_protected:D \tex_long:D \tex_gdef:D }
\tex_protected:D \tex_long:D \tex_gdef:D \cs_gset_protected:Npe
  { \tex_protected:D \tex_long:D \tex_xdef:D }
\tex_global:D \tex_let:D \cs_gset_protected:Npx \cs_gset_protected:Npe
```

(*End of definition for* `\cs_gset_nopar:Npn` *and others. These functions are documented on page 18.*)

\cs_set_nopar:Npn
\cs_set_nopar:Npe
\cs_set_nopar:Npx
\cs_set:Npn
\cs_set:Npe
\cs_set:Npx
\cs_set_protected_nopar:Npn
\cs_set_protected_nopar:Npe
\cs_set_protected_nopar:Npx
\cs_set_protected:Npn
\cs_set_protected:Npe
\cs_set_protected:Npx

Local versions of the above functions.

```
\tex_global:D \tex_let:D \cs_set_nopar:Npn          \tex_def:D
\tex_global:D \tex_let:D \cs_set_nopar:Npe          \tex_edef:D
\tex_global:D \tex_let:D \cs_set_nopar:Npx          \tex_edef:D
\cs_gset_protected:Npn \cs_set:Npn
  { \tex_long:D \tex_def:D }
\cs_gset_protected:Npn \cs_set:Npe
  { \tex_long:D \tex_edef:D }
\tex_global:D \tex_let:D \cs_set:Npx \cs_set:Npe
\cs_gset_protected:Npn \cs_set_protected_nopar:Npn
  { \tex_protected:D \tex_def:D }
\cs_gset_protected:Npn \cs_set_protected_nopar:Npe
  { \tex_protected:D \tex_edef:D }
\tex_global:D \tex_let:D \cs_set_protected_nopar:Npx \cs_set_protected_nopar:Npe
\cs_gset_protected:Npn \cs_set_protected:Npn
  { \tex_protected:D \tex_long:D \tex_def:D }
\cs_gset_protected:Npn \cs_set_protected:Npe
  { \tex_protected:D \tex_long:D \tex_edef:D }
\tex_global:D \tex_let:D \cs_set_protected:Npx \cs_set_protected:Npe
```

(*End of definition for* `\cs_set_nopar:Npn` *and others. These functions are documented on page 17.*)

42.4 Selecting tokens

```
⟨@@=exp⟩
```

\l__exp_internal_tl

Scratch token list variable for l3expan, used by `\use:x`, used in defining conditionals. We don't use `tl` methods because l3basics is loaded earlier.

```
\cs_gset_nopar:Npn \l__exp_internal_tl { }
```

(*End of definition for* `\l__exp_internal_tl`*.*)

\use:c

This macro grabs its argument and returns a csname from it.

```
\cs_gset:Npn \use:c #1 { \cs:w #1 \cs_end: }
```

(*End of definition for* `\use:c`*. This function is documented on page 22.*)

\use:x Fully expands its argument and passes it to the input stream. Uses the reserved \l__-exp_internal_tl which we've set up above.

```
\cs_gset_protected:Npn \use:x #1
  {
    \cs_set_nopar:Npx \l__exp_internal_tl {#1}
    \l__exp_internal_tl
  }
```

(*End of definition for* \use:x.)

```
⟨@@=use⟩
```

\use:e

```
\cs_gset:Npn \use:e #1 { \tex_expanded:D {#1} }
```

(*End of definition for* \use:e. *This function is documented on page 27.*)

```
⟨@@=exp⟩
```

\use:n
\use:nn
\use:nnn
\use:nnnn

These macros grab their arguments and return them back to the input (with outer braces removed).

```
\cs_gset:Npn \use:n    #1       {#1}
\cs_gset:Npn \use:nn   #1#2     {#1#2}
\cs_gset:Npn \use:nnn  #1#2#3   {#1#2#3}
\cs_gset:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}
```

(*End of definition for* \use:n *and others. These functions are documented on page 25.*)

\use_i:nn
\use_ii:nn

The equivalent to $\LaTeX 2_\varepsilon$'s \@firstoftwo and \@secondoftwo.

```
\cs_gset:Npn \use_i:nn  #1#2 {#1}
\cs_gset:Npn \use_ii:nn #1#2 {#2}
```

(*End of definition for* \use_i:nn *and* \use_ii:nn. *These functions are documented on page 26.*)

\use_i:nnn
\use_ii:nnn
\use_iii:nnn
\use_i:nnnn
\use_ii:nnnn
\use_iii:nnnn
\use_iv:nnnn
\use_i:nnnnn
\use_ii:nnnnn
\use_iii:nnnnn
\use_iv:nnnnn
\use_v:nnnnn
\use_i:nnnnnn
\use_ii:nnnnnn
\use_iii:nnnnnn
\use_iv:nnnnnn
\use_v:nnnnnn
\use_vi:nnnnnn
\use_i:nnnnnnn
\use_ii:nnnnnnn
\use_iii:nnnnnnn
\use_iv:nnnnnnn
\use_v:nnnnnnn
\use_vi:nnnnnnn
\use_vii:nnnnnnn
\use_i:nnnnnnnn
\use_ii:nnnnnnnn
\use_iii:nnnnnnnn
\use_iv:nnnnnnnn
\use_v:nnnnnnnn

We also need something for picking up arguments from a longer list.

```
\cs_gset:Npn \use_i:nnn     #1#2#3 {#1}
\cs_gset:Npn \use_ii:nnn    #1#2#3 {#2}
\cs_gset:Npn \use_iii:nnn   #1#2#3 {#3}
\cs_gset:Npn \use_i:nnnn    #1#2#3#4 {#1}
\cs_gset:Npn \use_ii:nnnn   #1#2#3#4 {#2}
\cs_gset:Npn \use_iii:nnnn  #1#2#3#4 {#3}
\cs_gset:Npn \use_iv:nnnn   #1#2#3#4 {#4}
\cs_gset:Npn \use_i:nnnnn    #1#2#3#4#5 {#1}
\cs_gset:Npn \use_ii:nnnnn   #1#2#3#4#5 {#2}
\cs_gset:Npn \use_iii:nnnnn  #1#2#3#4#5 {#3}
\cs_gset:Npn \use_iv:nnnnn   #1#2#3#4#5 {#4}
\cs_gset:Npn \use_v:nnnnn    #1#2#3#4#5 {#5}
\cs_gset:Npn \use_i:nnnnnn    #1#2#3#4#5#6 {#1}
\cs_gset:Npn \use_ii:nnnnnn   #1#2#3#4#5#6 {#2}
\cs_gset:Npn \use_iii:nnnnnn  #1#2#3#4#5#6 {#3}
\cs_gset:Npn \use_iv:nnnnnn   #1#2#3#4#5#6 {#4}
\cs_gset:Npn \use_v:nnnnnn    #1#2#3#4#5#6 {#5}
\cs_gset:Npn \use_vi:nnnnnn   #1#2#3#4#5#6 {#6}
\cs_gset:Npn \use_i:nnnnnnn    #1#2#3#4#5#6#7 {#1}
\cs_gset:Npn \use_ii:nnnnnnn   #1#2#3#4#5#6#7 {#2}
\cs_gset:Npn \use_iii:nnnnnnn  #1#2#3#4#5#6#7 {#3}
```

```
\cs_gset:Npn \use_iv:nnnnnnn  #1#2#3#4#5#6#7 {#4}
\cs_gset:Npn \use_v:nnnnnnn   #1#2#3#4#5#6#7 {#5}
\cs_gset:Npn \use_vi:nnnnnnn  #1#2#3#4#5#6#7 {#6}
\cs_gset:Npn \use_vii:nnnnnnn #1#2#3#4#5#6#7 {#7}
\cs_gset:Npn \use_i:nnnnnnnn    #1#2#3#4#5#6#7#8 {#1}
\cs_gset:Npn \use_ii:nnnnnnnn   #1#2#3#4#5#6#7#8 {#2}
\cs_gset:Npn \use_iii:nnnnnnnn  #1#2#3#4#5#6#7#8 {#3}
\cs_gset:Npn \use_iv:nnnnnnnn   #1#2#3#4#5#6#7#8 {#4}
\cs_gset:Npn \use_v:nnnnnnnn    #1#2#3#4#5#6#7#8 {#5}
\cs_gset:Npn \use_vi:nnnnnnnn   #1#2#3#4#5#6#7#8 {#6}
\cs_gset:Npn \use_vii:nnnnnnnn  #1#2#3#4#5#6#7#8 {#7}
\cs_gset:Npn \use_viii:nnnnnnnn #1#2#3#4#5#6#7#8 {#8}
\cs_gset:Npn \use_i:nnnnnnnnn    #1#2#3#4#5#6#7#8#9 {#1}
\cs_gset:Npn \use_ii:nnnnnnnnn   #1#2#3#4#5#6#7#8#9 {#2}
\cs_gset:Npn \use_iii:nnnnnnnnn  #1#2#3#4#5#6#7#8#9 {#3}
\cs_gset:Npn \use_iv:nnnnnnnnn   #1#2#3#4#5#6#7#8#9 {#4}
\cs_gset:Npn \use_v:nnnnnnnnn    #1#2#3#4#5#6#7#8#9 {#5}
\cs_gset:Npn \use_vi:nnnnnnnnn   #1#2#3#4#5#6#7#8#9 {#6}
\cs_gset:Npn \use_vii:nnnnnnnnn  #1#2#3#4#5#6#7#8#9 {#7}
\cs_gset:Npn \use_viii:nnnnnnnnn #1#2#3#4#5#6#7#8#9 {#8}
\cs_gset:Npn \use_ix:nnnnnnnnn   #1#2#3#4#5#6#7#8#9 {#9}
```

(*End of definition for* `\use_i:nnn` *and others. These functions are documented on page 26.*)

\use_i_ii:nnn

```
\cs_gset:Npn \use_i_ii:nnn #1#2#3 {#1#2}
```

(*End of definition for* `\use_i_ii:nnn`*. This function is documented on page 27.*)

\use_ii_i:nn

```
\cs_gset:Npn \use_ii_i:nn #1#2 { #2 #1 }
```

(*End of definition for* `\use_ii_i:nn`*. This function is documented on page 27.*)

\use_none_delimit_by_q_nil:w
\use_none_delimit_by_q_stop:w
\use_none_delimit_by_q_recursion_stop:w

Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_recursion_stop`, respectively.

```
\cs_gset:Npn \use_none_delimit_by_q_nil:w  #1 \q_nil  { }
\cs_gset:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
\cs_gset:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop { }
```

(*End of definition for* `\use_none_delimit_by_q_nil:w`*,* `\use_none_delimit_by_q_stop:w`*, and* `\use_none_delimit_by_q_recursion_stop:w`*. These functions are documented on page 27.*)

\use_i_delimit_by_q_nil:nw
\use_i_delimit_by_q_stop:nw
\use_i_delimit_by_q_recursion_stop:nw

Same as above but execute first argument after gobbling. Very useful when you need to skip the rest of a mapping sequence but want an easy way to control what should be expanded next.

```
\cs_gset:Npn \use_i_delimit_by_q_nil:nw  #1#2 \q_nil  {#1}
\cs_gset:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
\cs_gset:Npn \use_i_delimit_by_q_recursion_stop:nw
  #1#2 \q_recursion_stop {#1}
```

(*End of definition for* `\use_i_delimit_by_q_nil:nw`*,* `\use_i_delimit_by_q_stop:nw`*, and* `\use_i_delimit_by_q_recursion_stop:nw`*. These functions are documented on page 28.*)

42.5 Gobbling tokens from input

\use_none:n
\use_none:nn
\use_none:nnn
\use_none:nnnn
\use_none:nnnnn
\use_none:nnnnnn
\use_none:nnnnnnn
\use_none:nnnnnnnn
\use_none:nnnnnnnnn

To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of `n`'s following the `:` in the name. Although we could define functions to remove ten arguments or more using separate calls of `\use_none:nnnnn`, this is very non-intuitive to the programmer who will assume that expanding such a function once takes care of gobbling all the tokens in one go.

```
\cs_gset:Npn \use_none:n         #1                 { }
\cs_gset:Npn \use_none:nn        #1#2               { }
\cs_gset:Npn \use_none:nnn       #1#2#3             { }
\cs_gset:Npn \use_none:nnnn      #1#2#3#4           { }
\cs_gset:Npn \use_none:nnnnn     #1#2#3#4#5         { }
\cs_gset:Npn \use_none:nnnnnn    #1#2#3#4#5#6       { }
\cs_gset:Npn \use_none:nnnnnnn   #1#2#3#4#5#6#7     { }
\cs_gset:Npn \use_none:nnnnnnnn  #1#2#3#4#5#6#7#8   { }
\cs_gset:Npn \use_none:nnnnnnnnn #1#2#3#4#5#6#7#8#9 { }
```

(*End of definition for* `\use_none:n` *and others. These functions are documented on page 27.*)

42.6 Debugging and patching later definitions

```
⟨@@=debug⟩
```

__kernel_if_debug:TF

A more meaningful test of whether debugging is enabled than messing up with guards. We can also more easily change the logic in one place then. This is needed primarily for deprecations.

```
\cs_gset_protected:Npn \__kernel_if_debug:TF #1#2 {#2}
```

(*End of definition for* `\__kernel_if_debug:TF`.)

\debug_on:n
\debug_off:n

Stubs.

```
\cs_gset_protected:Npn \debug_on:n #1
  {
    \sys_load_debug:
    \cs_if_exist:NT \__debug_all_on:
      { \debug_on:n {#1} }
  }
\cs_gset_protected:Npn \debug_off:n #1
  {
    \sys_load_debug:
    \cs_if_exist:NT \__debug_all_on:
      { \debug_off:n {#1} }
  }
```

(*End of definition for* `\debug_on:n` *and* `\debug_off:n`. *These functions are documented on page 31.*)

\debug_suspend:
\debug_resume:

```
\cs_gset_protected:Npn \debug_suspend: { }
\cs_gset_protected:Npn \debug_resume: { }
```

(*End of definition for* `\debug_suspend:` *and* `\debug_resume:`. *These functions are documented on page 31.*)

__kernel_deprecation_code:nn
\g__debug_deprecation_on_tl
\g__debug_deprecation_off_tl

Make deprecated commands throw errors if the user requests it. This relies on two token lists, filled up in l3deprecation.

```
\cs_gset_nopar:Npn \g__debug_deprecation_on_tl { }
\cs_gset_nopar:Npn \g__debug_deprecation_off_tl { }
\cs_gset_protected:Npn \__kernel_deprecation_code:nn #1#2
  {
    \tl_gput_right:Nn \g__debug_deprecation_on_tl {#1}
    \tl_gput_right:Nn \g__debug_deprecation_off_tl {#2}
  }
```

(*End of definition for* `\__kernel_deprecation_code:nn`*,* `\g__debug_deprecation_on_tl`*, and* `\g__debug_deprecation_off_tl`*.*)

42.7 Conditional processing and definitions

```
⟨@@=prg⟩
```

Underneath any predicate function (`_p`) or other conditional forms (`TF`, etc.) is a built-in logic saying that it after all of the testing and processing must return the ⟨*state*⟩ this leaves TeX in. Therefore, a simple user interface could be something like

```
\if_meaning:w #1#2
  \prg_return_true:
\else:
  \if_meaning:w #1#3
    \prg_return_true:
  \else:
    \prg_return_false:
  \fi:
\fi:
```

Usually, a TeX programmer would have to insert a number of `\exp_after:wN`s to ensure the state value is returned at exactly the point where the last conditional is finished. However, that obscures the code and forces the TeX programmer to prove that he/she knows the $2^n - 1$ table. We therefore provide the simpler interface.

\prg_return_true:
\prg_return_false:

The idea here is that `\exp:w` expands fully any `\else:` and `\fi:` that are waiting to be discarded, before reaching the `\exp_end:` which leaves an empty expansion. The code can then leave either the first or second argument in the input stream. This means that all of the branching code has to contain at least two tokens: see how the logical tests are actually implemented to see this.

```
\cs_gset:Npn \prg_return_true:
  { \exp_after:wN \use_i:nn  \exp:w }
\cs_gset:Npn \prg_return_false:
  { \exp_after:wN \use_ii:nn \exp:w}
```

An extended state space could be implemented by including a more elaborate function in place of `\use_i:nn`/`\use_ii:nn`. Provided two arguments are absorbed then the code would work.

(*End of definition for* `\prg_return_true:` *and* `\prg_return_false:`*. These functions are documented on page* *66**.*)

__prg_use_none_delimit_by_q_recursion_stop:w

Private version of \use_none_delimit_by_q_recursion_stop:w.

```
\cs_gset:Npn \__prg_use_none_delimit_by_q_recursion_stop:w
  #1 \q__prg_recursion_stop { }
```

(*End of definition for* __prg_use_none_delimit_by_q_recursion_stop:w.)

\prg_set_conditional:Npnn
\prg_gset_conditional:Npnn
\prg_new_conditional:Npnn
\prg_set_protected_conditional:Npnn
\prg_gset_protected_conditional:Npnn
\prg_new_protected_conditional:Npnn
__prg_generate_conditional_parm:NNNpnn

The user functions for the types using parameter text from the programmer. The various functions only differ by which function is used for the assignment. For those Npnn type functions, we must grab the parameter text, reading everything up to a left brace before continuing. Then split the base function into name and signature, and feed {⟨*name*⟩} {⟨*signature*⟩} ⟨boolean⟩ {⟨*set or new*⟩} {⟨*maybe protected*⟩} {⟨*parameters*⟩} {TF,...} {⟨*code*⟩} to the auxiliary function responsible for defining all conditionals. Note that e stands for expandable and p for protected.

```
\cs_gset_protected:Npn \prg_set_conditional:Npnn
  { \__prg_generate_conditional_parm:NNNpnn \cs_set:Npn e }
\cs_gset_protected:Npn \prg_gset_conditional:Npnn
  { \__prg_generate_conditional_parm:NNNpnn \cs_gset:Npn e }
\cs_gset_protected:Npn \prg_new_conditional:Npnn
  { \__prg_generate_conditional_parm:NNNpnn \cs_new:Npn e }
\cs_gset_protected:Npn \prg_set_protected_conditional:Npnn
  { \__prg_generate_conditional_parm:NNNpnn \cs_set_protected:Npn p }
\cs_gset_protected:Npn \prg_gset_protected_conditional:Npnn
  { \__prg_generate_conditional_parm:NNNpnn \cs_gset_protected:Npn p }
\cs_gset_protected:Npn \prg_new_protected_conditional:Npnn
  { \__prg_generate_conditional_parm:NNNpnn \cs_new_protected:Npn p }
\cs_gset_protected:Npn \__prg_generate_conditional_parm:NNNpnn #1#2#3#4#
  {
    \use:e
      {
        \__prg_generate_conditional:nnNNNnnn
          \cs_split_function:N #3
      }
      #1 #2 {#4}
  }
```

(*End of definition for* \prg_set_conditional:Npnn *and others. These functions are documented on page 64.*)

\prg_set_conditional:Nnn
\prg_gset_conditional:Nnn
\prg_new_conditional:Nnn
\prg_set_protected_conditional:Nnn
\prg_gset_protected_conditional:Nnn
\prg_new_protected_conditional:Nnn
__prg_generate_conditional_count:NNNnn
__prg_generate_conditional_count:nnNNNnn

The user functions for the types automatically inserting the correct parameter text based on the signature. The various functions only differ by which function is used for the assignment. Split the base function into name and signature. The second auxiliary generates the parameter text from the number of letters in the signature. Then feed {⟨*name*⟩} {⟨*signature*⟩} ⟨boolean⟩ {⟨*set or new*⟩} {⟨*maybe protected*⟩} {⟨*parameters*⟩} {TF,...} {⟨*code*⟩} to the auxiliary function responsible for defining all conditionals. If the ⟨*signature*⟩ has more than 9 letters, the definition is aborted since TeX macros have at most 9 arguments. The erroneous case where the function name contains no colon is captured later.

```
\cs_gset_protected:Npn \prg_set_conditional:Nnn
  { \__prg_generate_conditional_count:NNNnn \cs_set:Npn e }
\cs_gset_protected:Npn \prg_gset_conditional:Nnn
  { \__prg_generate_conditional_count:NNNnn \cs_set:Npn e }
\cs_gset_protected:Npn \prg_new_conditional:Nnn
  { \__prg_generate_conditional_count:NNNnn \cs_new:Npn e }
```

```
\cs_gset_protected:Npn \prg_set_protected_conditional:Nnn
  { \__prg_generate_conditional_count:NNNnn \cs_set_protected:Npn p }
\cs_gset_protected:Npn \prg_gset_protected_conditional:Nnn
  { \__prg_generate_conditional_count:NNNnn \cs_gset_protected:Npn p }
\cs_gset_protected:Npn \prg_new_protected_conditional:Nnn
  { \__prg_generate_conditional_count:NNNnn \cs_new_protected:Npn p }
\cs_gset_protected:Npn \__prg_generate_conditional_count:NNNnn #1#2#3
  {
    \use:e
      {
        \__prg_generate_conditional_count:nnNNNnn
        \cs_split_function:N #3
      }
      #1 #2
  }
\cs_gset_protected:Npn \__prg_generate_conditional_count:nnNNNnn #1#2#3#4#5
  {
    \__kernel_cs_parm_from_arg_count:nnF
      { \__prg_generate_conditional:nnNNNnnn {#1} {#2} #3 #4 #5 }
      { \tl_count:n {#2} }
      {
        \msg_error:nnee { kernel } { bad-number-of-arguments }
          { \token_to_str:c { #1 : #2 } }
          { \tl_count:n {#2} }
        \use_none:nn
      }
  }
```

(*End of definition for* `\prg_set_conditional:Nnn` *and others. These functions are documented on page* *64*.)

`\__prg_generate_conditional:nnNNNnnn`
`\__prg_generate_conditional:NNnnnnNw`
`\__prg_generate_conditional_test:w`
`\__prg_generate_conditional_fast:nw`

The workhorse here is going through a list of desired forms, *i.e.*, `p`, `TF`, `T` and `F`. The first three arguments come from splitting up the base form of the conditional, which gives the name, signature and a boolean to signal whether or not there was a colon in the name. In the absence of a colon, we throw an error and don't define any conditional. The fourth and fifth arguments build up the defining function. The sixth is the parameters to use (possibly empty), the seventh is the list of forms to define, the eighth is the replacement text which we will augment when defining the forms. The use of `\tl_to_str:n` makes the later loop more robust.

A large number of our low-level conditionals look like ⟨*code*⟩ `\prg_return_true: \else: \prg_return_false: \fi:` so we optimize this special case by calling `\__prg_generate_conditional_fast:nw` {⟨*code*⟩}. This passes `\use_i:nn` instead of `\use_i_ii:nnn` to functions such as `\__prg_generate_p_form:wNNnnnnN`.

```
\cs_gset_protected:Npn \__prg_generate_conditional:nnNNNnnn #1#2#3#4#5#6#7#8
  {
    \if_meaning:w \c_false_bool #3
      \msg_error:nne { kernel } { missing-colon }
        { \token_to_str:c {#1} }
      \exp_after:wN \use_none:nn
    \fi:
    \use:e
      {
        \exp_not:N \__prg_generate_conditional:NNnnnnNw
```

```
        \exp_not:n { #4 #5 {#1} {#2} {#6} }
        \__prg_generate_conditional_test:w
          #8 \s__prg_mark
            \__prg_generate_conditional_fast:nw
          \prg_return_true: \else: \prg_return_false: \fi: \s__prg_mark
            \use_none:n
        \exp_not:n { {#8} \use_i_ii:nnn }
        \tl_to_str:n {#7}
        \exp_not:n { , \q__prg_recursion_tail , \q__prg_recursion_stop }
      }
  }
\cs_gset:Npn \__prg_generate_conditional_test:w
    #1 \prg_return_true: \else: \prg_return_false: \fi: \s__prg_mark #2
  { #2 {#1} }
\cs_gset:Npn \__prg_generate_conditional_fast:nw #1#2 \exp_not:n #3
  { \exp_not:n { {#1} \use_i:nn } }
```

Looping through the list of desired forms. First are six arguments and seventh is the form. Use the form to call the correct type. If the form does not exist, the `\use:c` construction results in `\relax`, and the error message is displayed (unless the form is empty, to allow for `{T, , F}`), then `\use_none:nnnnnnnn` cleans up. Otherwise, the error message is removed by the variant form.

```
\cs_gset_protected:Npn \__prg_generate_conditional:NNnnnnNw #1#2#3#4#5#6#7#8 ,
  {
    \if_meaning:w \q__prg_recursion_tail #8
      \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
    \fi:
    \use:c { __prg_generate_ #8 _form:wNNnnnnN }
        \tl_if_empty:nF {#8}
          {
            \msg_error:nnee
              { kernel } { conditional-form-unknown }
              {#8} { \token_to_str:c { #3 : #4 } }
          }
        \use_none:nnnnnnnn
      \s__prg_stop
      #1 #2 {#3} {#4} {#5} {#6} #7
    \__prg_generate_conditional:NNnnnnNw #1 #2 {#3} {#4} {#5} {#6} #7
  }
```

(*End of definition for `\__prg_generate_conditional:nnNNNnnn` and others.*)

`\__prg_generate_p_form:wNNnnnnN`
`\__prg_generate_TF_form:wNNnnnnN`
`\__prg_generate_T_form:wNNnnnnN`
`\__prg_generate_F_form:wNNnnnnN`
`\__prg_p_true:w`
`\__prg_T_true:w`
`\__prg_F_true:w`
`\__prg_TF_true:w`

How to generate the various forms. Those functions take the following arguments: 1: junk, 2: `\cs_set:Npn` or similar, 3: `p` (for protected conditionals) or `e`, 4: function name, 5: signature, 6: parameter text, 7: replacement (possibly trimmed by `\__prg_generate_conditional_fast:nw`), 8: `\use_i_ii:nnn` or `\use_i:nn` (for "fast" conditionals). Remember that the logic-returning functions expect two arguments to be present after `\exp_end:`: notice the construction of the different variants relies on this, and that the `TF` and `F` variants will be slightly faster than the `T` version. The `p` form is only valid for expandable tests, we check for that by making sure that the second argument is empty. For "fast" conditionals, `#7` has an extra `\if_...`. To optimize a bit further we don't use `\exp_after:wN \use_ii:nnn` and similar but instead use `\__prg_TF_true:w` and similar to swap out the macro after `\fi:`. It would be a tiny bit faster if we directly grabbed the T and F arguments there, but if those are actually missing, the recovery from the runaway argument would not insert `\fi:` back, messing up nesting of conditionals.

```
\cs_gset_protected:Npn \__prg_generate_p_form:wNNnnnnN
    #1 \s__prg_stop #2#3#4#5#6#7#8
  {
    \if_meaning:w e #3
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
      {
        #8
          { \exp_args:Nc #2 { #4 _p: #5 } #6 }
          { { #7 \exp_end: \c_true_bool \c_false_bool } }
          { #7 \__prg_p_true:w \fi: \c_false_bool }
      }
      {
        \msg_error:nne { kernel } { protected-predicate }
          { \token_to_str:c { #4 _p: #5 } }
      }
  }
\cs_gset_protected:Npn \__prg_generate_T_form:wNNnnnnN
    #1 \s__prg_stop #2#3#4#5#6#7#8
  {
    #8
      { \exp_args:Nc #2 { #4 : #5 T } #6 }
      { { #7 \exp_end: \use:n \use_none:n } }
      { #7 \__prg_T_true:w \fi: \use_none:n }
  }
\cs_gset_protected:Npn \__prg_generate_F_form:wNNnnnnN
    #1 \s__prg_stop #2#3#4#5#6#7#8
  {
    #8
      { \exp_args:Nc #2 { #4 : #5 F } #6 }
      { { #7 \exp_end: { } } }
      { #7 \__prg_F_true:w \fi: \use:n }
  }
\cs_gset_protected:Npn \__prg_generate_TF_form:wNNnnnnN
    #1 \s__prg_stop #2#3#4#5#6#7#8
  {
    #8
      { \exp_args:Nc #2 { #4 : #5 TF } #6 }
      { { #7 \exp_end: } }
      { #7 \__prg_TF_true:w \fi: \use_ii:nn }
  }
\cs_gset:Npn \__prg_p_true:w  \fi: \c_false_bool { \fi: \c_true_bool }
\cs_gset:Npn \__prg_T_true:w  \fi: \use_none:n   { \fi: \use:n }
\cs_gset:Npn \__prg_F_true:w  \fi: \use:n        { \fi: \use_none:n }
\cs_gset:Npn \__prg_TF_true:w \fi: \use_ii:nn    { \fi: \use_i:nn }
```

(*End of definition for* `\__prg_generate_p_form:wNNnnnnN` *and others.*)

`\prg_set_eq_conditional:NNn`
`\prg_gset_eq_conditional:NNn`
`\prg_new_eq_conditional:NNn`
`\__prg_set_eq_conditional:NNNn`

The setting-equal functions. Split both functions and feed {⟨*name*$_1$⟩} {⟨*signature*$_1$⟩} ⟨boolean$_1$⟩ {⟨*name*$_2$⟩} {⟨*signature*$_2$⟩} ⟨boolean$_2$⟩ ⟨copying function⟩ ⟨conditions⟩ ,

`\q__prg_recursion_tail , \q__prg_recursion_stop` to a first auxiliary.

```
\cs_gset_protected:Npn \prg_set_eq_conditional:NNn
  { \__prg_set_eq_conditional:NNNn \cs_set_eq:cc }
\cs_gset_protected:Npn \prg_gset_eq_conditional:NNn
  { \__prg_set_eq_conditional:NNNn \cs_gset_eq:cc }
\cs_gset_protected:Npn \prg_new_eq_conditional:NNn
  { \__prg_set_eq_conditional:NNNn \cs_new_eq:cc }
\cs_gset_protected:Npn \__prg_set_eq_conditional:NNNn #1#2#3#4
  {
    \use:e
      {
        \exp_not:N \__prg_set_eq_conditional:nnNnnNNw
          \cs_split_function:N #2
          \cs_split_function:N #3
          \exp_not:N #1
          \tl_to_str:n {#4}
          \exp_not:n { , \q__prg_recursion_tail , \q__prg_recursion_stop }
      }
  }
```

(*End of definition for* `\prg_set_eq_conditional:NNn` *and others. These functions are documented on page 66.*)

__prg_set_eq_conditional:nnNnnNNw
__prg_set_eq_conditional_loop:nnnnNw
__prg_set_eq_conditional_p_form:nnn
__prg_set_eq_conditional_TF_form:nnn
__prg_set_eq_conditional_T_form:nnn
__prg_set_eq_conditional_F_form:nnn

Split the function to be defined, and setup a manual clist loop over argument `#6` of the first auxiliary. The second auxiliary receives twice three arguments coming from splitting the function to be defined and the function to copy. Make sure that both functions contained a colon, otherwise we don't know how to build conditionals, hence abort. Call the looping macro, with arguments `{`⟨*name*$_1$⟩`}` `{`⟨*signature*$_1$⟩`}` `{`⟨*name*$_2$⟩`}` `{`⟨*signature*$_2$⟩`}` ⟨`copying function`⟩ and followed by the comma list. At each step in the loop, make sure that the conditional form we copy is defined, and copy it, otherwise abort.

```
\cs_gset_protected:Npn \__prg_set_eq_conditional:nnNnnNNw #1#2#3#4#5#6
  {
    \if_meaning:w \c_false_bool #3
      \msg_error:nne { kernel } { missing-colon }
        { \token_to_str:c {#1} }
      \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
    \fi:
    \if_meaning:w \c_false_bool #6
      \msg_error:nne { kernel } { missing-colon }
        { \token_to_str:c {#4} }
      \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
    \fi:
    \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#4} {#5}
  }
\cs_gset_protected:Npn \__prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 ,
  {
    \if_meaning:w \q__prg_recursion_tail #6
      \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
    \fi:
    \use:c { __prg_set_eq_conditional_ #6 _form:wNnnnn }
        \tl_if_empty:nF {#6}
          {
            \msg_error:nnee
              { kernel } { conditional-form-unknown }
```

```
            {#6} { \token_to_str:c { #1 : #2 } }
        }
      \use_none:nnnnnn
    \s__prg_stop
    #5 {#1} {#2} {#3} {#4}
  \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
}
\cs_gset:Npn \__prg_set_eq_conditional_p_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
  { #2 { #3 _p : #4    }    { #5 _p : #6    } }
\cs_gset:Npn \__prg_set_eq_conditional_TF_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
  { #2 { #3    : #4 TF }    { #5    : #6 TF } }
\cs_gset:Npn \__prg_set_eq_conditional_T_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
  { #2 { #3    : #4 T  }    { #5    : #6 T  } }
\cs_gset:Npn \__prg_set_eq_conditional_F_form:wNnnnn #1 \s__prg_stop #2#3#4#5#6
  { #2 { #3    : #4  F }    { #5    : #6  F } }
```

(*End of definition for* `\__prg_set_eq_conditional:nnNnnNNw` *and others.*)

All that is left is to define the canonical boolean true and false. I think Michael originated the idea of expandable boolean tests. At first these were supposed to expand into either `TT` or `TF` to be tested using `\if:w` but this was later changed to `00` and `01`, so they could be used in logical operations. Later again they were changed to being numerical constants with values of 1 for true and 0 for false. We need this from the get-go.

`\c_true_bool`
`\c_false_bool`

Here are the canonical boolean values.

```
\tex_global:D \tex_chardef:D \c_true_bool  = 1 ~
\tex_global:D \tex_chardef:D \c_false_bool = 0 ~
```

(*End of definition for* `\c_true_bool` *and* `\c_false_bool`*. These variables are documented on page 68.*)

42.8 Dissecting a control sequence

```
⟨@@=cs⟩
```

`\__cs_count_signature:N` ⋆

`\__cs_count_signature:N` ⟨*function*⟩

Splits the ⟨*function*⟩ into the ⟨*name*⟩ (*i.e.* the part before the colon) and the ⟨*signature*⟩ (*i.e.* after the colon). The ⟨*number*⟩ of tokens in the ⟨*signature*⟩ is then left in the input stream. If there was no ⟨*signature*⟩ then the result is the marker value -1.

`\__cs_tmp:w`

Function used for various short-term usages, for instance defining functions whose definition involves tokens which are hard to insert normally (spaces, characters with category other).

`\cs_to_str:N`
`\__cs_to_str:N`
`\__cs_to_str:w`

This converts a control sequence into the character string of its name, removing the leading escape character. This turns out to be a non-trivial matter as there a different cases:

- The usual case of a printable escape character;
- the case of a non-printable escape characters, e.g., when the value of the `\escapechar` is negative;

- when the escape character is a space.

One approach to solve this is to test how many tokens result from `\token_to_str:N \a`. If there are two tokens, then the escape character is printable, while if it is non-printable then only one is present.

However, there is an additional complication: the control sequence itself may start with a space. Clearly that should *not* be lost in the process of converting to a string. So the approach adopted is a little more intricate still. When the escape character is printable, `\token_to_str:N␣\␣` yields the escape character itself and a space. The character codes are different, thus the `\if:w` test is false, and TeX reads `\__cs_to_str:N` after turning the following control sequence into a string; this auxiliary removes the escape character, and stops the expansion of the initial `\tex_romannumeral:D`. The second case is that the escape character is not printable. Then the `\if:w` test is unfinished after reading a the space from `\token_to_str:N␣\␣`, and the auxiliary `\__cs_to_str:w` is expanded, feeding `-` as a second character for the test; the test is false, and TeX skips to `\fi:`, then performs `\token_to_str:N`, and stops the `\tex_romannumeral:D` with `\c_zero_int`. The last case is that the escape character is itself a space. In this case, the `\if:w` test is true, and the auxiliary `\__cs_to_str:w` comes into play, inserting `-\int_value:w`, which expands `\c_zero_int` to the character `0`. The initial `\tex_romannumeral:D` then sees `0`, which is not a terminated number, followed by the escape character, a space, which is removed, terminating the expansion of `\tex_romannumeral:D`. In all three cases, `\cs_to_str:N` takes two expansion steps to be fully expanded.

```
\cs_gset:Npn \cs_to_str:N
  {
```

We implement the expansion scheme using `\tex_romannumeral:D` terminating it with `\c_zero_int` rather than using `\exp:w` and `\exp_end:` as we normally do. The reason is that the code heavily depends on terminating the expansion with `\c_zero_int` so we make this dependency explicit.

```
    \tex_romannumeral:D
      \if:w \token_to_str:N \ \__cs_to_str:w \fi:
      \exp_after:wN \__cs_to_str:N \token_to_str:N
  }
\cs_gset:Npn \__cs_to_str:N #1 { \c_zero_int }
\cs_gset:Npn \__cs_to_str:w #1 \__cs_to_str:N
  { - \int_value:w \fi: \exp_after:wN \c_zero_int }
```

If speed is a concern we could use `\csstring` in LuaTeX. For the empty csname that primitive gives an empty result while the current `\cs_to_str:N` gives incorrect results in all engines (this is impossible to fix without huge performance hit).

(*End of definition for* `\cs_to_str:N`*,* `\__cs_to_str:N`*, and* `\__cs_to_str:w`*. This function is documented on page* *23**.*)

`\cs_split_function:N`
`\__cs_split_function_auxi:w`
`\__cs_split_function_auxii:w`

This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean ⟨*true*⟩ or ⟨*false*⟩ is returned with ⟨*true*⟩ for when there is a colon in the function and ⟨*false*⟩ if there is not.

First ensure that we actually get a properly evaluated string by expanding `\cs_to_str:N` twice. If the function contained a colon, the auxiliary takes as `#1` the function name, delimited by the first colon, then the signature `#2`, delimited by `\s__cs_mark`, then `\c_true_bool` as `#3`, and `#4` cleans up until `\s__cs_stop`. Otherwise, the `#1` contains the function name and `\s__cs_mark \c_true_bool`, `#2` is empty, `#3` is `\c_false_bool`, and #4 cleans up. The second auxiliary trims the trailing `\s__cs_mark` from the function name if present (that is, if the original function had no colon).

```
\cs_gset_protected:Npn \__cs_tmp:w #1
  {
    \cs_gset:Npn \cs_split_function:N ##1
      {
        \exp_after:wN \exp_after:wN \exp_after:wN
        \__cs_split_function_auxi:w
          \cs_to_str:N ##1 \s__cs_mark \c_true_bool
          #1 \s__cs_mark \c_false_bool \s__cs_stop
      }
    \cs_gset:Npn \__cs_split_function_auxi:w
        ##1 #1 ##2 \s__cs_mark ##3##4 \s__cs_stop
      { \__cs_split_function_auxii:w ##1 \s__cs_mark \s__cs_stop {##2} ##3 }
    \cs_gset:Npn \__cs_split_function_auxii:w ##1 \s__cs_mark ##2 \s__cs_stop
      { {##1} }
  }
\exp_after:wN \__cs_tmp:w \token_to_str:N :
```

(*End of definition for* `\cs_split_function:N`*,* `\__cs_split_function_auxi:w`*, and* `\__cs_split_function_auxii:w`*. This function is documented on page 23.*)

42.9 Exist or free

A control sequence is said to *exist* (to be used) if has an entry in the hash table and its meaning is different from the primitive `\relax` token. A control sequence is said to be *free* (to be defined) if it does not already exist.

`\cs_if_exist_p:N`
`\cs_if_exist_p:c`
`\cs_if_exist:NTF`
`\cs_if_exist:cTF`
`\__cs_if_exist_c_aux:`
`\__cs_if_exist_c_aux:w`

Two versions for checking existence. For the `N` form we firstly check for `\scan_stop:` and then if it is in the hash table. There is no problem when inputting something like `\else:` or `\fi:` as TeX will only ever skip input in case the token tested against is `\scan_stop:`.

In both the `N` and `c` form we use the way `\prg_set_conditional:Npnn` optimizes the conditionals to negate the tests using `\else:` (the `\else:` in the top level functions will be removed by the optimization, and this usage of `\else:` will be fine).

```
\prg_gset_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \scan_stop:
      \use_i:nnnn
    \else:
    \fi:
    \if_cs_exist:N #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

For the `c` form we firstly check if it is in the hash table and then for `\scan_stop:` so that we do not add it to the hash table unless it was already there. Here we have to be careful as the text to be skipped if the first test is false may contain tokens that disturb the scanner. Therefore, we ensure that the second test is performed after the first one has concluded completely.

```
\cs_if_exist:NTF \tex_lastnamedcs:D
  {
    \prg_gset_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
      {
        \if_cs_exist:w #1 \cs_end:
          \__cs_if_exist_c_aux:
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      }
    \cs_gset:Npn \__cs_if_exist_c_aux:
      { \fi: \exp_after:wN \if_meaning:w \tex_lastnamedcs:D \scan_stop: \else: }
  }
  {
    \prg_gset_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
      {
        \if_cs_exist:w #1 \cs_end:
          \__cs_if_exist_c_aux:w
        \fi:
        \use_none:n {#1}
        \if_false:
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      }
    \cs_gset:Npn \__cs_if_exist_c_aux:w \fi: \use_none:n #1 \if_false:
      { \fi: \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop: \else: }
  }
```

(*End of definition for* `\cs_if_exist:NTF`*,* `\__cs_if_exist_c_aux:`*, and* `\__cs_if_exist_c_aux:w`*. This function is documented on page 29.*)

`\cs_if_free_p:N`
`\cs_if_free_p:c`
`\cs_if_free:NTF`
`\cs_if_free:cTF`

The logical reversal of the above.

```
\prg_gset_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
  {
    \if_cs_exist:N #1
    \else:
      \use_none:nnnn
    \fi:
    \if_meaning:w #1 \scan_stop:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_if_exist:NTF \tex_lastnamedcs:D
  {
    \prg_gset_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
      {
        \if_cs_exist:w #1 \cs_end:
          \__cs_if_free_c_aux:w
        \fi:
```

```
        \if_true:
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      }
    \cs_gset:Npn \__cs_if_free_c_aux:w \fi: \if_true:
      { \fi: \exp_after:wN \if_meaning:w \tex_lastnamedcs:D \scan_stop: }
  }
  {
    \prg_gset_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
      {
        \if_cs_exist:w #1 \cs_end:
          \__cs_if_free_c_aux:w
        \fi:
        \use_none:n {#1}
        \if_true:
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      }
    \cs_gset:Npn \__cs_if_free_c_aux:w \fi: \use_none:n #1 \if_true:
      { \fi: \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop: }
  }
```

(*End of definition for* `\cs_if_free:NTF`. *This function is documented on page 29.*)

`\cs_if_exist_use:N`
`\cs_if_exist_use:c`
`\cs_if_exist_use:NTF`
`\cs_if_exist_use:cTF`
`\__cs_if_exist_use_aux:w`
`\__cs_if_exist_use_aux:Nnn`

The `\cs_if_exist_use:...` functions cannot be implemented as conditionals because the true branch must leave both the control sequence itself and the true code in the input stream. For the `c` variants, we are careful not to put the control sequence in the hash table if it does not exist. If available we use the `\lastnamedcs` primitive.

```
\cs_gset:Npn \cs_if_exist_use:NTF #1#2
  { \cs_if_exist:NTF #1 { #1 #2 } }
\cs_gset:Npn \cs_if_exist_use:NF #1
  { \cs_if_exist:NTF #1 #1 }
\cs_gset:Npn \cs_if_exist_use:NT #1 #2
  { \cs_if_exist:NT #1 { #1 #2 } }
\cs_gset:Npn \cs_if_exist_use:N #1
  { \cs_if_exist:NT #1 #1 }
\cs_if_exist:NTF \tex_lastnamedcs:D
  {
    \cs_gset:Npn \cs_if_exist_use:cTF #1
      {
        \if_cs_exist:w #1 \cs_end:
          \__cs_if_exist_use_aux:w
        \fi:
        \use_ii:nn
      }
    \cs_gset:Npn \__cs_if_exist_use_aux:w \fi: \use_ii:nn
      { \fi: \exp_after:wN \__cs_if_exist_use_aux:Nnn \tex_lastnamedcs:D }
  }
  {
    \cs_gset:Npn \cs_if_exist_use:cTF #1
```

```
      {
        \if_cs_exist:w #1 \cs_end:
          \__cs_if_exist_use_aux:w
        \fi:
        \use_iii:nnn {#1}
      }
    \cs_gset:Npn \__cs_if_exist_use_aux:w \fi: \use_iii:nnn #1
      { \fi: \exp_after:wN \__cs_if_exist_use_aux:Nnn \cs:w #1 \cs_end: }
  }
\cs_gset:Npn \__cs_if_exist_use_aux:Nnn #1#2
  {
    \if_meaning:w #1 \scan_stop:
      \exp_after:wN \use_iii:nnn
    \fi:
    \use_i:nn { #1 #2 }
  }
\cs_gset:Npn \cs_if_exist_use:cF #1
  { \cs_if_exist_use:cTF {#1} {} }
\cs_gset:Npn \cs_if_exist_use:cT #1#2
  { \cs_if_exist_use:cTF {#1} {#2} {} }
\cs_gset:Npn \cs_if_exist_use:c #1
  { \cs_if_exist_use:cTF {#1} {} {} }
```

(*End of definition for* `\cs_if_exist_use:NTF`*,* `\__cs_if_exist_use_aux:w`*, and* `\__cs_if_exist_use_aux:Nnn`*. This function is documented on page 22.*)

42.10 Preliminaries for new functions

We provide two kinds of functions that can be used to define control sequences. On the one hand we have functions that check if their argument doesn't already exist, they are called `\..._new`. The second type of defining functions doesn't check if the argument is already defined.

Before we can define them, we need some auxiliary macros that allow us to generate error messages. The next few definitions here are only temporary, they will be redefined later on.

`\msg_error:nnee`
`\msg_error:nne`
`\msg_error:nn`

If an internal error occurs before LaTeX3 has loaded l3msg then the code should issue a usable if terse error message and halt. This can only happen if a coding error is made by the team, so this is a reasonable response. Setting the `\newlinechar` is needed, to turn `^^J` into a proper line break in plain TeX.

```
\cs_gset_protected:Npn \msg_error:nnee #1#2#3#4
  {
    \tex_newlinechar:D = `\^^J \scan_stop:
    \tex_errmessage:D
      {
        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^^J
        Argh,~internal~LaTeX3~error! ^^J ^^J
        Module ~ #1 , ~ message~name~"#2": ^^J
        Arguments~'#3'~and~'#4' ^^J ^^J
        This~is~one~for~The~LaTeX3~Project:~bailing~out
      }
    \tex_end:D
  }
```

```
\cs_gset_protected:Npn \msg_error:nne #1#2#3
  { \msg_error:nnee {#1} {#2} {#3} { } }
\cs_gset_protected:Npn \msg_error:nn #1#2
  { \msg_error:nnee {#1} {#2} { } { } }
```

(*End of definition for* `\msg_error:nnnn`. *This function is documented on page 85.*)

`\msg_line_context:` Another one from l3msg which will be altered later.

```
\cs_gset:Npn \msg_line_context:
  { on~line~ \tex_the:D \tex_inputlineno:D }
```

(*End of definition for* `\msg_line_context:`. *This function is documented on page 83.*)

`\iow_log:e` `\iow_term:e` We define a routine to write only to the log file. And a similar one for writing to both the log file and the terminal. These will be redefined later by l3file.

```
\cs_gset_protected:Npn \iow_log:e
  { \tex_immediate:D \tex_write:D -1 }
\cs_gset_protected:Npn \iow_term:e
  { \tex_immediate:D \tex_write:D 16 }
```

(*End of definition for* `\iow_log:n`. *This function is documented on page 97.*)

`\__kernel_chk_if_free_cs:N` `\__kernel_chk_if_free_cs:c` This command is called by `\cs_new_nopar:Npn` and `\cs_new_eq:NN` *etc.* to make sure that the argument sequence is not already in use. If it is, an error is signalled. It checks if ⟨*csname*⟩ is undefined or `\scan_stop:`. Otherwise an error message is issued. We have to make sure we don't put the argument into the conditional processing since it may be an `\if...` type function!

```
\cs_gset_protected:Npn \__kernel_chk_if_free_cs:N #1
  {
    \cs_if_free:NF #1
      {
        \msg_error:nnee { kernel } { command-already-defined }
          { \token_to_str:N #1 } { \token_to_meaning:N #1 }
      }
  }
\cs_gset_protected:Npn \__kernel_chk_if_free_cs:c
  { \exp_args:Nc \__kernel_chk_if_free_cs:N }
```

(*End of definition for* `\__kernel_chk_if_free_cs:N`.)

42.11 Defining new functions

```
⟨@@=cs⟩
```

`\cs_new_nopar:Npn` `\cs_new_nopar:Npe` `\cs_new_nopar:Npx` `\cs_new:Npn` `\cs_new:Npe` `\cs_new:Npx` `\cs_new_protected_nopar:Npn` `\cs_new_protected_nopar:Npe` `\cs_new_protected_nopar:Npx` `\cs_new_protected:Npn` `\cs_new_protected:Npe` `\cs_new_protected:Npx` `\__cs_tmp:w` Function which check that the control sequence is free before defining it.

```
\cs_set:Npn \__cs_tmp:w #1#2
  {
    \cs_gset_protected:Npn #1 ##1
      {
        \__kernel_chk_if_free_cs:N ##1
        #2 ##1
      }
  }
\__cs_tmp:w \cs_new_nopar:Npn           \cs_gset_nopar:Npn
```

```
\__cs_tmp:w \cs_new_nopar:Npe           \cs_gset_nopar:Npe
\__cs_tmp:w \cs_new_nopar:Npx           \cs_gset_nopar:Npx
\__cs_tmp:w \cs_new:Npn                 \cs_gset:Npn
\__cs_tmp:w \cs_new:Npe                 \cs_gset:Npe
\__cs_tmp:w \cs_new:Npx                 \cs_gset:Npx
\__cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
\__cs_tmp:w \cs_new_protected_nopar:Npe \cs_gset_protected_nopar:Npe
\__cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
\__cs_tmp:w \cs_new_protected:Npn       \cs_gset_protected:Npn
\__cs_tmp:w \cs_new_protected:Npe       \cs_gset_protected:Npe
\__cs_tmp:w \cs_new_protected:Npx       \cs_gset_protected:Npx
```

(*End of definition for* `\cs_new_nopar:Npn` *and others. These functions are documented on page 16.*)

`\cs_set_nopar:cpn`
`\cs_set_nopar:cpe`
`\cs_set_nopar:cpx`
`\cs_gset_nopar:cpn`
`\cs_gset_nopar:cpe`
`\cs_gset_nopar:cpx`
`\cs_new_nopar:cpn`
`\cs_new_nopar:cpe`
`\cs_new_nopar:cpx`

Like `\cs_set_nopar:Npn` and `\cs_new_nopar:Npn`, except that the first argument consists of the sequence of characters that should be used to form the name of the desired control sequence (the `c` stands for csname argument, see the expansion module). Global versions are also provided.

`\cs_set_nopar:cpn`⟨*string*⟩⟨*rep-text*⟩ turns ⟨*string*⟩ into a csname and then assigns ⟨*rep-text*⟩ to it by using `\cs_set_nopar:Npn`. This means that there might be a parameter string between the two arguments.

```
\cs_set:Npn \__cs_tmp:w #1#2
  { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
\__cs_tmp:w \cs_set_nopar:cpn  \cs_set_nopar:Npn
\__cs_tmp:w \cs_set_nopar:cpe  \cs_set_nopar:Npe
\__cs_tmp:w \cs_set_nopar:cpx  \cs_set_nopar:Npx
\__cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
\__cs_tmp:w \cs_gset_nopar:cpe \cs_gset_nopar:Npe
\__cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
\__cs_tmp:w \cs_new_nopar:cpn  \cs_new_nopar:Npn
\__cs_tmp:w \cs_new_nopar:cpe  \cs_new_nopar:Npe
\__cs_tmp:w \cs_new_nopar:cpx  \cs_new_nopar:Npx
```

(*End of definition for* `\cs_set_nopar:Npn`. *This function is documented on page 17.*)

`\cs_set:cpn`
`\cs_set:cpe`
`\cs_set:cpx`
`\cs_gset:cpn`
`\cs_gset:cpe`
`\cs_gset:cpx`
`\cs_new:cpn`
`\cs_new:cpe`
`\cs_new:cpx`

Variants of the `\cs_set:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```
\__cs_tmp:w \cs_set:cpn  \cs_set:Npn
\__cs_tmp:w \cs_set:cpe  \cs_set:Npe
\__cs_tmp:w \cs_set:cpx  \cs_set:Npx
\__cs_tmp:w \cs_gset:cpn \cs_gset:Npn
\__cs_tmp:w \cs_gset:cpe \cs_gset:Npe
\__cs_tmp:w \cs_gset:cpx \cs_gset:Npx
\__cs_tmp:w \cs_new:cpn  \cs_new:Npn
\__cs_tmp:w \cs_new:cpe  \cs_new:Npe
\__cs_tmp:w \cs_new:cpx  \cs_new:Npx
```

(*End of definition for* `\cs_set:Npn`. *This function is documented on page 17.*)

`\cs_set_protected_nopar:cpn`
`\cs_set_protected_nopar:cpe`
`\cs_set_protected_nopar:cpx`
`\cs_gset_protected_nopar:cpn`
`\cs_gset_protected_nopar:cpe`
`\cs_gset_protected_nopar:cpx`
`\cs_new_protected_nopar:cpn`
`\cs_new_protected_nopar:cpe`
`\cs_new_protected_nopar:cpx`

Variants of the `\cs_set_protected_nopar:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```
\__cs_tmp:w \cs_set_protected_nopar:cpn  \cs_set_protected_nopar:Npn
\__cs_tmp:w \cs_set_protected_nopar:cpe  \cs_set_protected_nopar:Npe
\__cs_tmp:w \cs_set_protected_nopar:cpx  \cs_set_protected_nopar:Npx
```

```
\__cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
\__cs_tmp:w \cs_gset_protected_nopar:cpe \cs_gset_protected_nopar:Npe
\__cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
\__cs_tmp:w \cs_new_protected_nopar:cpn  \cs_new_protected_nopar:Npn
\__cs_tmp:w \cs_new_protected_nopar:cpe  \cs_new_protected_nopar:Npe
\__cs_tmp:w \cs_new_protected_nopar:cpx  \cs_new_protected_nopar:Npx
```

(*End of definition for* `\cs_set_protected_nopar:Npn`. *This function is documented on page 17.*)

`\cs_set_protected:cpn`
`\cs_set_protected:cpe`
`\cs_set_protected:cpx`
`\cs_gset_protected:cpn`
`\cs_gset_protected:cpe`
`\cs_gset_protected:cpx`
`\cs_new_protected:cpn`
`\cs_new_protected:cpe`
`\cs_new_protected:cpx`

Variants of the `\cs_set_protected:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```
\__cs_tmp:w \cs_set_protected:cpn  \cs_set_protected:Npn
\__cs_tmp:w \cs_set_protected:cpe  \cs_set_protected:Npe
\__cs_tmp:w \cs_set_protected:cpx  \cs_set_protected:Npx
\__cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
\__cs_tmp:w \cs_gset_protected:cpe \cs_gset_protected:Npe
\__cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
\__cs_tmp:w \cs_new_protected:cpn  \cs_new_protected:Npn
\__cs_tmp:w \cs_new_protected:cpe  \cs_new_protected:Npe
\__cs_tmp:w \cs_new_protected:cpx  \cs_new_protected:Npx
```

(*End of definition for* `\cs_set_protected:Npn`. *This function is documented on page 17.*)

42.12 Copying definitions

`\cs_set_eq:NN`
`\cs_set_eq:cN`
`\cs_set_eq:Nc`
`\cs_set_eq:cc`
`\cs_gset_eq:NN`
`\cs_gset_eq:cN`
`\cs_gset_eq:Nc`
`\cs_gset_eq:cc`
`\cs_new_eq:NN`
`\cs_new_eq:cN`
`\cs_new_eq:Nc`
`\cs_new_eq:cc`

These macros allow us to copy the definition of a control sequence to another control sequence.

The `=` sign allows us to define funny char tokens like `=` itself or ␣ with this function. For the definition of `\c_space_char{~}` to work we need the `~` after the `=`.

`\cs_set_eq:NN` is long to avoid problems with a literal argument of `\par`. While `\cs_new_eq:NN` will probably never be correct with a first argument of `\par`, define it long in order to throw an “already defined” error rather than “runaway argument”.

```
\cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }
\cs_new_protected:Npn \cs_set_eq:cN { \exp_args:Nc  \cs_set_eq:NN }
\cs_new_protected:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
\cs_new_protected:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
\cs_new_protected:Npn \cs_gset_eq:NN { \tex_global:D  \cs_set_eq:NN }
\cs_new_protected:Npn \cs_gset_eq:Nc { \exp_args:NNc  \cs_gset_eq:NN }
\cs_new_protected:Npn \cs_gset_eq:cN { \exp_args:Nc   \cs_gset_eq:NN }
\cs_new_protected:Npn \cs_gset_eq:cc { \exp_args:Ncc  \cs_gset_eq:NN }
\cs_new_protected:Npn \cs_new_eq:NN #1
  {
    \__kernel_chk_if_free_cs:N #1
    \tex_global:D \cs_set_eq:NN #1
  }
\cs_new_protected:Npn \cs_new_eq:cN { \exp_args:Nc  \cs_new_eq:NN }
\cs_new_protected:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
\cs_new_protected:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }
```

(*End of definition for* `\cs_set_eq:NN`, `\cs_gset_eq:NN`, *and* `\cs_new_eq:NN`. *These functions are documented on page 21.*)

42.13 Undefining functions

\cs_undefine:N
\cs_undefine:c

The following function is used to free the main memory from the definition of some function that isn't in use any longer. The `c` variant is careful not to add the control sequence to the hash table if it isn't there yet, and it also avoids nesting TeX conditionals in case `#1` is unbalanced in this matter. We optimize the case where the command exists by reducing as much as possible the tokens in the conditional.

```
\cs_new_protected:Npn \cs_undefine:N #1
  { \cs_gset_eq:NN #1 \tex_undefined:D }
\cs_new_protected:Npn \cs_undefine:c #1
  {
    \if_cs_exist:w #1 \cs_end:
    \else:
      \use_i:nnnn
    \fi:
    \exp_args:Nc \cs_undefine:N {#1}
  }
```

(*End of definition for* `\cs_undefine:N`. *This function is documented on page 21.*)

42.14 Generating parameter text from argument count

```
⟨@@=cs⟩
```

__kernel_cs_parm_from_arg_count:nnF
__cs_parm_from_arg_count_test:nnF

LaTeX3 provides shorthands to define control sequences and conditionals with a simple parameter text, derived directly from the signature, or more generally from knowing the number of arguments, between 0 and 9. This function expands to its first argument, untouched, followed by a brace group containing the parameter text `{#1...#`n`}`, where n is the result of evaluating the second argument (as described in `\int_eval:n`). If the second argument gives a result outside the range $[0, 9]$, the third argument is returned instead, normally an error message. Some of the functions use here are not defined yet, but will be defined before this function is called.

```
\cs_new_protected:Npn \__kernel_cs_parm_from_arg_count:nnF #1#2
  {
    \exp_args:Ne \__cs_parm_from_arg_count_test:nnF
      {
        \exp_after:wN \exp_not:n
        \if_case:w \int_eval:n {#2}
             { }
        \or: { ##1 }
        \or: { ##1##2 }
        \or: { ##1##2##3 }
        \or: { ##1##2##3##4 }
        \or: { ##1##2##3##4##5 }
        \or: { ##1##2##3##4##5##6 }
        \or: { ##1##2##3##4##5##6##7 }
        \or: { ##1##2##3##4##5##6##7##8 }
        \or: { ##1##2##3##4##5##6##7##8##9 }
        \else: { \c_false_bool }
        \fi:
      }
      {#1}
```

```
  }
\cs_new_protected:Npn \__cs_parm_from_arg_count_test:nnF #1#2
  {
    \if_meaning:w \c_false_bool #1
      \exp_after:wN \use_ii:nn
    \else:
      \exp_after:wN \use_i:nn
    \fi:
    { #2 {#1} }
  }
```

(*End of definition for* `\__kernel_cs_parm_from_arg_count:nnF` *and* `\__cs_parm_from_arg_count_test:nnF`.)

42.15 Defining functions from a given number of arguments

```
⟨@@=cs⟩
```

`\__cs_count_signature:N`
`\__cs_count_signature:c`
`\__cs_count_signature:n`
`\__cs_count_signature:nnN`

Counting the number of tokens in the signature, *i.e.*, the number of arguments the function should take. Since this is not used in any time-critical function, we simply use `\tl_count:n` if there is a signature, otherwise -1 arguments to signal an error. We need a variant form right away.

```
\cs_new:Npn \__cs_count_signature:N #1
  { \exp_args:Nf \__cs_count_signature:n { \cs_split_function:N #1 } }
\cs_new:Npn \__cs_count_signature:n #1
  { \int_eval:n { \__cs_count_signature:nnN #1 } }
\cs_new:Npn \__cs_count_signature:nnN #1#2#3
  {
    \if_meaning:w \c_true_bool #3
      \tl_count:n {#2}
    \else:
      -1
    \fi:
  }
\cs_new:Npn \__cs_count_signature:c
  { \exp_args:Nc \__cs_count_signature:N }
```

(*End of definition for* `\__cs_count_signature:N`*,* `\__cs_count_signature:n`*, and* `\__cs_count_signature:nnN`.)

`\cs_generate_from_arg_count:NNnn`
`\cs_generate_from_arg_count:cNnn`
`\cs_generate_from_arg_count:Ncnn`

We provide a constructor function for defining functions with a given number of arguments. For this we need to choose the correct parameter text and then use that when defining. Since TeX supports from zero to nine arguments, we use a simple switch to choose the correct parameter text, ensuring the result is returned after finishing the conditional. If it is not between zero and nine, we throw an error.

1: function to define, 2: with what to define it, 3: the number of args it requires and 4: the replacement text

```
\cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
  {
    \__kernel_cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
      {
        \msg_error:nnee { kernel } { bad-number-of-arguments }
          { \token_to_str:N #1 } { \int_eval:n {#3} }
```

```
        \use_none:n
      }
      {#4}
  }
```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```
\cs_new_protected:Npn \cs_generate_from_arg_count:cNnn
  { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
\cs_new_protected:Npn \cs_generate_from_arg_count:Ncnn
  { \exp_args:NNc \cs_generate_from_arg_count:NNnn }
```

(*End of definition for* `\cs_generate_from_arg_count:NNnn`*. This function is documented on page 20.*)

42.16 Using the signature to define functions

```
⟨@@=cs⟩
```

We can now combine some of the tools we have to provide a simple interface for defining functions, where the number of arguments is read from the signature. For instance, `\cs_set:Nn \foo_bar:nn {#1,#2}`.

`\cs_set:Nn`
`\cs_set:Ne`
`\cs_set:Nx`
`\cs_set_nopar:Nn`
`\cs_set_nopar:Ne`
`\cs_set_nopar:Nx`
`\cs_set_protected:Nn`
`\cs_set_protected:Ne`
`\cs_set_protected:Nx`
`\cs_set_protected_nopar:Nn`
`\cs_set_protected_nopar:Ne`
`\cs_set_protected_nopar:Nx`
`\cs_gset:Nn`
`\cs_gset:Ne`
`\cs_gset:Nx`
`\cs_gset_nopar:Nn`
`\cs_gset_nopar:Ne`
`\cs_gset_nopar:Nx`
`\cs_gset_protected:Nn`
`\cs_gset_protected:Ne`
`\cs_gset_protected:Nx`
`\cs_gset_protected_nopar:Nn`
`\cs_gset_protected_nopar:Ne`
`\cs_gset_protected_nopar:Nx`
`\cs_new:Nn`
`\cs_new:Ne`
`\cs_new:Nx`
`\cs_new_nopar:Nn`
`\cs_new_nopar:Ne`
`\cs_new_nopar:Nx`
`\cs_new_protected:Nn`
`\cs_new_protected:Ne`
`\cs_new_protected:Nx`
`\cs_new_protected_nopar:Nn`
`\cs_new_protected_nopar:Ne`
`\cs_new_protected_nopar:Nx`

We want to define `\cs_set:Nn` as

```
\cs_set_protected:Npn \cs_set:Nn #1#2
  {
    \cs_generate_from_arg_count:NNnn #1 \cs_set:Npn
      { \@@_count_signature:N #1 } {#2}
  }
```

In short, to define `\cs_set:Nn` we need just use `\cs_set:Npn`, everything else is the same for each variant. Therefore, we can make it simpler by temporarily defining a function to do this for us.

```
\cs_set:Npn \__cs_tmp:w #1#2#3
  {
    \cs_new_protected:cpx { cs_ #1 : #2 }
      {
        \exp_not:N \__cs_generate_from_signature:NNn
        \exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
      }
  }
\cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
  {
    \use:e
      {
        \__cs_generate_from_signature:nnNNNn
        \cs_split_function:N #2
      }
      #1 #2
  }
\cs_new_protected:Npn \__cs_generate_from_signature:nnNNNn #1#2#3#4#5#6
  {
    \bool_if:NTF #3
      {
        \cs_set_nopar:Npx \__cs_tmp:w
```

```
          { \tl_map_function:nN {#2} \__cs_generate_from_signature:n }
        \tl_if_empty:oF \__cs_tmp:w
          {
            \msg_error:nneee { kernel } { non-base-function }
              { \token_to_str:N #5 } {#2} { \__cs_tmp:w }
          }
        \cs_generate_from_arg_count:NNnn
          #5 #4 { \tl_count:n {#2} } {#6}
      }
      {
        \msg_error:nne { kernel } { missing-colon }
          { \token_to_str:N #5 }
      }
  }
\cs_new:Npn \__cs_generate_from_signature:n #1
  {
    \if:w n #1 \else: \if:w N #1 \else:
    \if:w T #1 \else: \if:w F #1 \else: #1 \fi: \fi: \fi: \fi:
  }
```

Then we define the 24 variants beginning with N.

```
\__cs_tmp:w { set }                   { Nn } { Npn }
\__cs_tmp:w { set }                   { Ne } { Npe }
\__cs_tmp:w { set }                   { Nx } { Npx }
\__cs_tmp:w { set_nopar }             { Nn } { Npn }
\__cs_tmp:w { set_nopar }             { Ne } { Npe }
\__cs_tmp:w { set_nopar }             { Nx } { Npx }
\__cs_tmp:w { set_protected }         { Nn } { Npn }
\__cs_tmp:w { set_protected }         { Ne } { Npe }
\__cs_tmp:w { set_protected }         { Nx } { Npx }
\__cs_tmp:w { set_protected_nopar }   { Nn } { Npn }
\__cs_tmp:w { set_protected_nopar }   { Ne } { Npe }
\__cs_tmp:w { set_protected_nopar }   { Nx } { Npx }
\__cs_tmp:w { gset }                  { Nn } { Npn }
\__cs_tmp:w { gset }                  { Ne } { Npe }
\__cs_tmp:w { gset }                  { Nx } { Npx }
\__cs_tmp:w { gset_nopar }            { Nn } { Npn }
\__cs_tmp:w { gset_nopar }            { Ne } { Npe }
\__cs_tmp:w { gset_nopar }            { Nx } { Npx }
\__cs_tmp:w { gset_protected }        { Nn } { Npn }
\__cs_tmp:w { gset_protected }        { Ne } { Npe }
\__cs_tmp:w { gset_protected }        { Nx } { Npx }
\__cs_tmp:w { gset_protected_nopar }  { Nn } { Npn }
\__cs_tmp:w { gset_protected_nopar }  { Ne } { Npe }
\__cs_tmp:w { gset_protected_nopar }  { Nx } { Npx }
\__cs_tmp:w { new }                   { Nn } { Npn }
\__cs_tmp:w { new }                   { Ne } { Npe }
\__cs_tmp:w { new }                   { Nx } { Npx }
\__cs_tmp:w { new_nopar }             { Nn } { Npn }
\__cs_tmp:w { new_nopar }             { Ne } { Npe }
\__cs_tmp:w { new_nopar }             { Nx } { Npx }
\__cs_tmp:w { new_protected }         { Nn } { Npn }
\__cs_tmp:w { new_protected }         { Ne } { Npe }
\__cs_tmp:w { new_protected }         { Nx } { Npx }
\__cs_tmp:w { new_protected_nopar }   { Nn } { Npn }
```

```
\__cs_tmp:w { new_protected_nopar }  { Ne } { Npe }
\__cs_tmp:w { new_protected_nopar }  { Nx } { Npx }
```

(*End of definition for* `\cs_set:Nn` *and others. These functions are documented on page 19.*)

`\cs_set:cn`
`\cs_set:ce`
`\cs_set:cx`
`\cs_set_nopar:cn`
`\cs_set_nopar:ce`
`\cs_set_nopar:cx`
`\cs_set_protected:cn`
`\cs_set_protected:ce`
`\cs_set_protected:cx`
`\cs_set_protected_nopar:cn`
`\cs_set_protected_nopar:ce`
`\cs_set_protected_nopar:cx`
`\cs_gset:cn`
`\cs_gset:ce`
`\cs_gset:cx`
`\cs_gset_nopar:cn`
`\cs_gset_nopar:ce`
`\cs_gset_nopar:cx`
`\cs_gset_protected:cn`
`\cs_gset_protected:ce`
`\cs_gset_protected:cx`
`\cs_gset_protected_nopar:cn`
`\cs_gset_protected_nopar:ce`
`\cs_gset_protected_nopar:cx`
`\cs_new:cn`
`\cs_new:ce`
`\cs_new:cx`
`\cs_new_nopar:cn`
`\cs_new_nopar:ce`
`\cs_new_nopar:cx`
`\cs_new_protected:cn`
`\cs_new_protected:ce`
`\cs_new_protected:cx`
`\cs_new_protected_nopar:cn`
`\cs_new_protected_nopar:ce`
`\cs_new_protected_nopar:cx`

The 24 `c` variants simply use `\exp_args:Nc`.

```
\cs_set:Npn \__cs_tmp:w #1#2
  {
    \cs_new_protected:cpx { cs_ #1 : c #2 }
      {
        \exp_not:N \exp_args:Nc
        \exp_after:wN \exp_not:N \cs:w cs_ #1 : N #2 \cs_end:
      }
  }
\__cs_tmp:w { set }                   { n }
\__cs_tmp:w { set }                   { e }
\__cs_tmp:w { set }                   { x }
\__cs_tmp:w { set_nopar }             { n }
\__cs_tmp:w { set_nopar }             { e }
\__cs_tmp:w { set_nopar }             { x }
\__cs_tmp:w { set_protected }         { n }
\__cs_tmp:w { set_protected }         { e }
\__cs_tmp:w { set_protected }         { x }
\__cs_tmp:w { set_protected_nopar }   { n }
\__cs_tmp:w { set_protected_nopar }   { e }
\__cs_tmp:w { set_protected_nopar }   { x }
\__cs_tmp:w { gset }                  { n }
\__cs_tmp:w { gset }                  { e }
\__cs_tmp:w { gset }                  { x }
\__cs_tmp:w { gset_nopar }            { n }
\__cs_tmp:w { gset_nopar }            { e }
\__cs_tmp:w { gset_nopar }            { x }
\__cs_tmp:w { gset_protected }        { n }
\__cs_tmp:w { gset_protected }        { e }
\__cs_tmp:w { gset_protected }        { x }
\__cs_tmp:w { gset_protected_nopar }  { n }
\__cs_tmp:w { gset_protected_nopar }  { e }
\__cs_tmp:w { gset_protected_nopar }  { x }
\__cs_tmp:w { new }                   { n }
\__cs_tmp:w { new }                   { e }
\__cs_tmp:w { new }                   { x }
\__cs_tmp:w { new_nopar }             { n }
\__cs_tmp:w { new_nopar }             { e }
\__cs_tmp:w { new_nopar }             { x }
\__cs_tmp:w { new_protected }         { n }
\__cs_tmp:w { new_protected }         { e }
\__cs_tmp:w { new_protected }         { x }
\__cs_tmp:w { new_protected_nopar }   { n }
\__cs_tmp:w { new_protected_nopar }   { e }
\__cs_tmp:w { new_protected_nopar }   { x }
```

(*End of definition for* `\cs_set:Nn`. *This function is documented on page 19.*)

42.17 Checking control sequence equality

\cs_if_eq_p:NN
\cs_if_eq_p:cN
\cs_if_eq_p:Nc
\cs_if_eq_p:cc
\cs_if_eq:NNTF
\cs_if_eq:cNTF
\cs_if_eq:NcTF
\cs_if_eq:ccTF

Check if two control sequences are identical.

```
\prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
  {
    \if_meaning:w #1#2
      \prg_return_true: \else: \prg_return_false: \fi:
  }
\cs_new:Npn \cs_if_eq_p:cN { \exp_args:Nc  \cs_if_eq_p:NN }
\cs_new:Npn \cs_if_eq:cNTF { \exp_args:Nc  \cs_if_eq:NNTF }
\cs_new:Npn \cs_if_eq:cNT  { \exp_args:Nc  \cs_if_eq:NNT }
\cs_new:Npn \cs_if_eq:cNF  { \exp_args:Nc  \cs_if_eq:NNF }
\cs_new:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
\cs_new:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
\cs_new:Npn \cs_if_eq:NcT  { \exp_args:NNc \cs_if_eq:NNT }
\cs_new:Npn \cs_if_eq:NcF  { \exp_args:NNc \cs_if_eq:NNF }
\cs_new:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
\cs_new:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
\cs_new:Npn \cs_if_eq:ccT  { \exp_args:Ncc \cs_if_eq:NNT }
\cs_new:Npn \cs_if_eq:ccF  { \exp_args:Ncc \cs_if_eq:NNF }
```

(*End of definition for* `\cs_if_eq:NNTF`. *This function is documented on page 29.*)

42.18 Diagnostic functions

```
⟨@@=kernel⟩
```

__kernel_chk_defined:NT

Error if the variable #1 is not defined.

```
\cs_new_protected:Npn \__kernel_chk_defined:NT #1#2
  {
    \cs_if_exist:NTF #1
      {#2}
      {
        \msg_error:nne { kernel } { variable-not-defined }
          { \token_to_str:N #1 }
      }
  }
```

(*End of definition for* `\__kernel_chk_defined:NT`.)

__kernel_register_show:N
__kernel_register_show:c
__kernel_register_log:N
__kernel_register_log:c
__kernel_register_show_aux:NN
__kernel_register_show_aux:nNN

Simply using the `\showthe` primitive does not allow for line-wrapping, so instead use `\tl_show:n` and `\tl_log:n` (defined in l3tl and that performs line-wrapping). This displays `>~`⟨*variable*⟩`=`⟨*value*⟩. We expand the value before-hand as otherwise some integers (such as `\currentgrouplevel` or `\currentgrouptype`) altered by the line-wrapping code would show wrong values.

```
\cs_new_protected:Npn \__kernel_register_show:N
  { \__kernel_register_show_aux:NN \tl_show:n }
\cs_new_protected:Npn \__kernel_register_show:c
  { \exp_args:Nc \__kernel_register_show:N }
\cs_new_protected:Npn \__kernel_register_log:N
  { \__kernel_register_show_aux:NN \tl_log:n }
\cs_new_protected:Npn \__kernel_register_log:c
  { \exp_args:Nc \__kernel_register_log:N }
```

```
\cs_new_protected:Npn \__kernel_register_show_aux:NN #1#2
  {
    \__kernel_chk_defined:NT #2
      {
        \exp_args:No \__kernel_register_show_aux:nNN
          { \tex_the:D #2 } #2 #1
      }
  }
\cs_new_protected:Npn \__kernel_register_show_aux:nNN #1#2#3
  { \exp_args:No #3 { \token_to_str:N #2 = #1 } }
```

(*End of definition for* `\__kernel_register_show:N` *and others.*)

`\cs_show:N`
`\cs_show:c`
`\cs_log:N`
`\cs_log:c`
`\__kernel_show:NN`

Some control sequences have a very long name or meaning. Thus, simply using TEX's primitive `\show` could lead to overlong lines. The output of this primitive is mimicked to some extent, then the re-built string is given to `\tl_show:n` or `\tl_log:n` for line-wrapping. We must expand the meaning before passing it to the wrapping code as otherwise we would wrongly see the definitions that are in place there. To get correct escape characters, set the `\escapechar` in a group; this also localizes the assignment performed by `e`-expansion. The `\cs_show:c` and `\cs_log:c` commands convert their argument to a control sequence within a group to avoid showing `\relax` for undefined control sequences.

```
\cs_new_protected:Npn \cs_show:N { \__kernel_show:NN \tl_show:n }
\cs_new_protected:Npn \cs_show:c
  { \group_begin: \exp_args:NNc \group_end: \cs_show:N }
\cs_new_protected:Npn \cs_log:N { \__kernel_show:NN \tl_log:n }
\cs_new_protected:Npn \cs_log:c
  { \group_begin: \exp_args:NNc \group_end: \cs_log:N }
\cs_new_protected:Npn \__kernel_show:NN #1#2
  {
    \group_begin:
      \int_set:Nn \tex_escapechar:D { `\\ }
      \exp_args:NNe
    \group_end:
    #1 { \token_to_str:N #2 = \cs_meaning:N #2 }
  }
```

(*End of definition for* `\cs_show:N`, `\cs_log:N`, *and* `\__kernel_show:NN`. *These functions are documented on page 21.*)

`\group_show_list:`
`\group_log_list:`
`\__kernel_group_show:NN`

Wrapper around `\showgroups`. Getting TEX to write to the log without interruption the run is done by altering the interaction mode.

```
\cs_new_protected:Npn \group_show_list:
  { \__kernel_group_show:NN \use_none:n 1 }
\cs_new_protected:Npn \group_log_list:
  { \__kernel_group_show:NN \int_gzero:N 0 }
\cs_new_protected:Npn \__kernel_group_show:NN #1#2
  {
    \use:e
      {
        #1 \tex_interactionmode:D
        \int_set:Nn \tex_tracingonline:D  {#2}
        \int_set:Nn \tex_errorcontextlines:D { -1 }
        \exp_not:N \exp_after:wN \scan_stop:
```

```
        \tex_showgroups:D
        \int_gset:Nn \tex_interactionmode:D
          { \int_use:N \tex_interactionmode:D }
        \int_set:Nn \tex_tracingonline:D
          { \int_use:N \tex_tracingonline:D }
        \int_set:Nn \tex_errorcontextlines:D
          { \int_use:N \tex_errorcontextlines:D }
      }
  }
```

(*End of definition for* `\group_show_list:` *,* `\group_log_list:` *, and* `\__kernel_group_show:NN`*. These functions are documented on page 15.*)

42.19 Decomposing a macro definition

`\cs_prefix_spec:N`
`\cs_parameter_spec:N`
`\cs_replacement_spec:N`
`\__kernel_prefix_arg_replacement:wN`

We sometimes want to test if a control sequence can be expanded to reveal a hidden value. However, we cannot just expand the macro blindly as it may have arguments and none might be present. Therefore we define these functions to pick either the prefix(es), the parameter specification, or the replacement text from a macro. All of this information is returned as characters with catcode 12. If the token in question isn't a macro, the token `\scan_stop:` is returned instead.

```
\use:e
  {
    \exp_not:n { \cs_new:Npn \__kernel_prefix_arg_replacement:wN #1 }
    \tl_to_str:n { macro : } \exp_not:n { #2 -> #3 \s__kernel_stop #4 }
  }
  { #4 {#1} {#2} {#3} }
\cs_new:Npn \cs_prefix_spec:N #1
  {
    \token_if_macro:NTF #1
      {
        \exp_after:wN \__kernel_prefix_arg_replacement:wN
          \token_to_meaning:N #1 \s__kernel_stop \use_i:nnn
      }
      { \scan_stop: }
  }
\cs_new:Npn \cs_parameter_spec:N #1
  {
    \token_if_macro:NTF #1
      {
        \exp_after:wN \__kernel_prefix_arg_replacement:wN
          \token_to_meaning:N #1 \s__kernel_stop \use_ii:nnn
      }
      { \scan_stop: }
  }
\cs_new:Npn \cs_replacement_spec:N #1
  {
    \token_if_macro:NTF #1
      {
        \exp_after:wN \__kernel_prefix_arg_replacement:wN
          \token_to_meaning:N #1 \s__kernel_stop \use_iii:nnn
      }
      { \scan_stop: }
```

```
  }
```

(*End of definition for* `\cs_prefix_spec:N` *and others. These functions are documented on page* 23.)

42.20 Doing nothing functions

`\prg_do_nothing:`

This does not fit anywhere else!

```
\cs_new:Npn \prg_do_nothing: { }
```

(*End of definition for* `\prg_do_nothing:`*. This function is documented on page* 14.)

42.21 Breaking out of mapping functions

```
⟨@@=prg⟩
```

`\prg_break_point:Nn`
`\prg_map_break:Nn`

In inline mappings, the nesting level must be reset at the end of the mapping, even when the user decides to break out. This is done by putting the code that must be performed as an argument of `\__prg_break_point:Nn`. The breaking functions are then defined to jump to that point and perform the argument of `\__prg_break_point:Nn`, before the user's code (if any). There is a check that we close the correct loop, otherwise we continue breaking.

```
\cs_new_eq:NN \prg_break_point:Nn \use_ii:nn
\cs_new:Npn \prg_map_break:Nn #1#2#3 \prg_break_point:Nn #4#5
  {
    #5
    \if_meaning:w #1 #4
      \exp_after:wN \use_iii:nnn
    \fi:
    \prg_map_break:Nn #1 {#2}
  }
```

(*End of definition for* `\prg_break_point:Nn` *and* `\prg_map_break:Nn`*. These functions are documented on page* 73.)

`\prg_break_point:`
`\prg_break:`
`\prg_break:n`

Very simple analogues of `\prg_break_point:Nn` and `\prg_map_break:Nn`, for use in fast short-term recursions which are not mappings, do not need to support nesting, and in which nothing has to be done at the end of the loop.

```
\cs_new_eq:NN \prg_break_point: \prg_do_nothing:
\cs_new:Npn \prg_break: #1 \prg_break_point: { }
\cs_new:Npn \prg_break:n #1#2 \prg_break_point: {#1}
```

(*End of definition for* `\prg_break_point:`, `\prg_break:`, *and* `\prg_break:n`*. These functions are documented on page* 74.)

42.22 Starting a paragraph

`\mode_leave_vertical:`

The approach here is different to that used by LaTeX 2_ε or plain TeX, which unbox a void box to force horizontal mode. That inserts the `\everypar` tokens *before* the re-inserted unboxing tokens. The approach here uses a protected macro, equivalent to the `\quitvmode` primitive. In vertical mode, the `\indent` primitive is inserted: this will switch to horizontal mode and insert `\everypar` tokens and nothing else. Unlike the

LaTeX 2$_\varepsilon$ version, the availability of ε-TeX means using a mode test can be done at for example the start of an `\halign`.

```
\cs_new_protected:Npn \mode_leave_vertical:
  {
    \if_mode_vertical:
      \exp_after:wN \tex_indent:D
    \fi:
  }
```

(*End of definition for* `\mode_leave_vertical:`. *This function is documented on page 31.*)

```
</package>
```

Chapter 43

l3expan implementation

```
⟨*package⟩
⟨@@=exp⟩
```

\l__exp_internal_tl The `\exp_` module has its private variable to temporarily store the result of `x`-type argument expansion. This is done to avoid interference with other functions using temporary variables.

(*End of definition for* `\l__exp_internal_tl`.)

\exp_after:wN
\exp_not:N
\exp_not:n

These are defined in l3basics, as they are needed "early". This is just a reminder of that fact!

(*End of definition for* `\exp_after:wN`*,* `\exp_not:N`*, and* `\exp_not:n`*. These functions are documented on page 40.*)

43.1 General expansion

In this section a general mechanism for defining functions that handle arguments is defined. These general expansion functions are expandable unless `x` is used. (Any version of `x` is going to have to use one of the LaTeX3 names for `\cs_set:Npx` at some point, and so is never going to be expandable.)

The definition of expansion functions with this technique happens in section 43.7. In section 43.2 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

\l__exp_internal_tl This scratch token list variable is defined in l3basics.

(*End of definition for* `\l__exp_internal_tl`.)

This code uses internal functions with names that start with `\::` to perform the expansions. All macros are `long` since the tokens undergoing expansion may be arbitrary user input.

An argument manipulator `\::`⟨*Z*⟩ always has signature `#1\:::#2#3` where `#1` holds the remaining argument manipulations to be performed, `\:::` serves as an end marker for the list of manipulations, `#2` is the carried over result of the previous expansion steps and `#3` is the argument about to be processed. One exception to this rule is `\::p`, which has to grab an argument delimited by a left brace.

__exp_arg_next:nnn
__exp_arg_next:Nnn

#1 is the result of an expansion step, #2 is the remaining argument manipulations and #3 is the current result of the expansion chain. This auxiliary function moves #1 back after #3 in the input stream and checks if any expansion is left to be done by calling #2. In by far the most cases we need to add a set of braces to the result of an argument manipulation so it is more effective to do it directly here. Actually, so far only the c of the final argument manipulation variants does not require a set of braces.

```
\cs_new:Npn \__exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }
\cs_new:Npn \__exp_arg_next:Nnn #1#2#3 { #2 \::: { #3 #1 } }
```

(*End of definition for* __exp_arg_next:nnn *and* __exp_arg_next:Nnn.)

\::: The end marker is just another name for the identity function.

```
\cs_new:Npn \::: #1 {#1}
```

(*End of definition for* \:::. *This function is documented on page 44.*)

\::n This function is used to skip an argument that doesn't need to be expanded.

```
\cs_new:Npn \::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(*End of definition for* \::n. *This function is documented on page 44.*)

\::N This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```
\cs_new:Npn \::N #1 \::: #2#3 { #1 \::: {#2#3} }
```

(*End of definition for* \::N. *This function is documented on page 44.*)

\::p This function is used to skip an argument that is delimited by a left brace and doesn't need to be expanded. It is not wrapped in braces in the result.

```
\cs_new:Npn \::p #1 \::: #2#3# { #1 \::: {#2#3} }
```

(*End of definition for* \::p. *This function is documented on page 44.*)

\::c This function is used to skip an argument that is turned into a control sequence without expansion.

```
\cs_new:Npn \::c #1 \::: #2#3
  { \exp_after:wN \__exp_arg_next:Nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(*End of definition for* \::c. *This function is documented on page 44.*)

\::o This function is used to expand an argument once.

```
\cs_new:Npn \::o #1 \::: #2#3
  { \exp_after:wN \__exp_arg_next:nnn \exp_after:wN {#3} {#1} {#2} }
```

(*End of definition for* \::o. *This function is documented on page 44.*)

\::e With the \expanded primitive available, just expand.

```
\cs_new:Npn \::e #1 \::: #2#3
  { \tex_expanded:D { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } } }
```

(*End of definition for* \::e. *This function is documented on page 44.*)

`\::f` `\exp_stop_f:` This function is used to expand a token list until the first unexpandable token is found. This is achieved through `\exp:w \exp_end_continue_f:w` that expands everything in its way following it. This scanning procedure is terminated once the expansion hits something non-expandable (if that is a space it is removed). We introduce `\exp_stop_f:` to mark such an end-of-expansion marker. For example, `f`-expanding `\cs_set_eq:Nc \aaa { b \l_tmpa_tl b }` where `\l_tmpa_tl` contains the characters `lur` gives `\tex_let:D \aaa = \blurb` which then turns out to start with the non-expandable token `\tex_let:D`. Since the expansion of `\exp:w \exp_end_continue_f:w` is empty, we wind up with a fully expanded list, only TeX has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `e` and `x` argument type.

```
\cs_new:Npn \::f #1 \::: #2#3
  {
    \exp_after:wN \__exp_arg_next:nnn
      \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
      {#1} {#2}
  }
\use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }
```

(*End of definition for* `\::f` *and* `\exp_stop_f:`. *These functions are documented on page 44.*)

`\::x` This function is used to expand an argument fully. We build in the expansion of `\__exp_arg_next:nnn`.

```
\cs_new_protected:Npn \::x #1 \::: #2#3
  {
    \cs_set_nopar:Npe \l__exp_internal_tl
      { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } }
    \l__exp_internal_tl
  }
```

(*End of definition for* `\::x`. *This function is documented on page 44.*)

`\::v` `\::V` These functions return the value of a register, i.e., one of `tl`, `clist`, `int`, `skip`, `dim`, `muskip`, or built-in TeX register. The `V` version expects a single token whereas `v` like `c` creates a csname from its argument given in braces and then evaluates it as if it was a `V`. The `\exp:w` sets off an expansion similar to an `f`-type expansion, which we terminate using `\exp_end:`. The argument is returned in braces.

```
\cs_new:Npn \::V #1 \::: #2#3
  {
    \exp_after:wN \__exp_arg_next:nnn
      \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
      {#1} {#2}
}
\cs_new:Npn \::v #1 \::: #2#3
  {
    \exp_after:wN \__exp_arg_next:nnn
      \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
      {#1} {#2}
  }
```

(*End of definition for* `\::v` *and* `\::V`. *These functions are documented on page 44.*)

`\__exp_eval_register:N`
`\__exp_eval_register:c`
`\__exp_eval_error_msg:w`

This function evaluates a register. Now a register might exist as one of two things: A parameter-less macro or a built-in TeX register such as `\count`. For the TeX registers we have to utilize a `\the` whereas for the macros we merely have to expand them once. The trick is to find out when to use `\the` and when not to. What we want here is to find out whether the token expands to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the token in question when it has been prefixed with `\exp_not:N` and the token itself. If it is a macro, the prefixed `\exp_not:N` temporarily turns it into the primitive `\scan_stop:`.

```
\cs_new:Npn \__exp_eval_register:N #1
  {
    \exp_after:wN \if_meaning:w \exp_not:N #1 #1
```

If the token was not a macro it may be a malformed variable from a `c` expansion in which case it is equal to the primitive `\scan_stop:`. In that case we throw an error. We could let TeX do it for us but that would result in the rather obscure

```
! You can't use '\relax' after \the.
```

which while quite true doesn't give many hints as to what actually went wrong. We provide something more sensible.

```
      \if_meaning:w \scan_stop: #1
        \__exp_eval_error_msg:w
      \fi:
```

The next bit requires some explanation. The function must be initiated by `\exp:w` and we want to terminate this expansion chain by inserting the `\exp_end:` token. However, we have to expand the register `#1` before we do that. If it is a TeX register, we need to execute the sequence `\exp_after:wN \exp_end: \tex_the:D #1` and if it is a macro we need to execute `\exp_after:wN \exp_end: #1`. We therefore issue the longer of the two sequences and if the register is a macro, we remove the `\tex_the:D`.

```
    \else:
      \exp_after:wN \use_i_ii:nnn
    \fi:
    \exp_after:wN \exp_end: \tex_the:D #1
  }
\cs_new:Npn \__exp_eval_register:c #1
  { \exp_after:wN \__exp_eval_register:N \cs:w #1 \cs_end: }
```

Clean up nicely, then call the undefined control sequence. The result is an error message looking like this:

```
! Undefined control sequence.
<argument> \LaTeX3 error:
                           Erroneous variable used!
l.55 \tl_set:Nv \l_tmpa_tl {undefined_tl}
```

```
\cs_new:Npn \__exp_eval_error_msg:w #1 \tex_the:D #2
  {
      \fi:
    \fi:
    \msg_expandable_error:nnn { kernel } { bad-variable } {#2}
    \exp_end:
  }
```

(*End of definition for* `\__exp_eval_register:N` *and* `\__exp_eval_error_msg:w`.)

43.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable.

\exp_args:Nc
\exp_args:cc

In l3basics.

(*End of definition for* \exp_args:Nc *and* \exp_args:cc. *These functions are documented on page 37.*)

\exp_args:NNc
\exp_args:Ncc
\exp_args:Nccc

Here are the functions that turn their argument into csnames but are expandable.

```
\cs_new:Npn \exp_args:NNc #1#2#3
  { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
\cs_new:Npn \exp_args:Ncc #1#2#3
  { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
\cs_new:Npn \exp_args:Nccc #1#2#3#4
  {
    \exp_after:wN #1
      \cs:w #2 \exp_after:wN \cs_end:
      \cs:w #3 \exp_after:wN \cs_end:
      \cs:w #4 \cs_end:
  }
```

(*End of definition for* \exp_args:NNc, \exp_args:Ncc, *and* \exp_args:Nccc. *These functions are documented on page 38.*)

\exp_args:No
\exp_args:NNo
\exp_args:NNNo

Those lovely runs of expansion!

```
\cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
\cs_new:Npn \exp_args:NNo #1#2#3
  { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
\cs_new:Npn \exp_args:NNNo #1#2#3#4
  { \exp_after:wN #1 \exp_after:wN#2 \exp_after:wN #3 \exp_after:wN {#4} }
```

(*End of definition for* \exp_args:No, \exp_args:NNo, *and* \exp_args:NNNo. *These functions are documented on page 37.*)

\exp_args:Ne

When the \expanded primitive is available, use it.

```
\cs_new:Npn \exp_args:Ne #1#2
  { \exp_after:wN #1 \tex_expanded:D { {#2} } }
```

(*End of definition for* \exp_args:Ne. *This function is documented on page 37.*)

\exp_args:Nf
\exp_args:NV
\exp_args:Nv

```
\cs_new:Npn \exp_args:Nf #1#2
  { \exp_after:wN #1 \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } }
\cs_new:Npn \exp_args:Nv #1#2
  {
    \exp_after:wN #1 \exp_after:wN
      { \exp:w \__exp_eval_register:c {#2} }
  }
\cs_new:Npn \exp_args:NV #1#2
  {
    \exp_after:wN #1 \exp_after:wN
      { \exp:w \__exp_eval_register:N #2 }
  }
```

(*End of definition for* \exp_args:Nf, \exp_args:NV, *and* \exp_args:Nv. *These functions are documented on page 37.*)

\exp_args:NNV
\exp_args:NNv
\exp_args:NNe
\exp_args:NNf
\exp_args:Nco
\exp_args:NcV
\exp_args:Ncv
\exp_args:Ncf
\exp_args:NVV

Some more hand-tuned function with three arguments. If we forced that an o argument always has braces, we could implement \exp_args:Nco with less tokens and only two arguments.

```
\cs_new:Npn \exp_args:NNV #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
  }
\cs_new:Npn \exp_args:NNv #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
  }
\cs_new:Npn \exp_args:NNe #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \tex_expanded:D { {#3} }
  }
\cs_new:Npn \exp_args:NNf #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
  }
\cs_new:Npn \exp_args:Nco #1#2#3
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN {#3}
  }
\cs_new:Npn \exp_args:NcV #1#2#3
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
  }
\cs_new:Npn \exp_args:Ncv #1#2#3
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
  }
\cs_new:Npn \exp_args:Ncf #1#2#3
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
  }
\cs_new:Npn \exp_args:NVV #1#2#3
  {
    \exp_after:wN #1
```

```
    \exp_after:wN { \exp:w \exp_after:wN
      \__exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
    \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
  }
```

(*End of definition for* `\exp_args:NNV` *and others. These functions are documented on page* 38.)

`\exp_args:NNNV`
`\exp_args:NNNv`
`\exp_args:NNNe`
`\exp_args:NcNc`
`\exp_args:NcNo`
`\exp_args:Ncco`

A few more that we can hand-tune.

```
\cs_new:Npn \exp_args:NNNV #1#2#3#4
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN { \exp:w \__exp_eval_register:N #4 }
  }
\cs_new:Npn \exp_args:NNNv #1#2#3#4
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN { \exp:w \__exp_eval_register:c {#4} }
  }
\cs_new:Npn \exp_args:NNNe #1#2#3#4
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \tex_expanded:D { {#4} }
  }
\cs_new:Npn \exp_args:NcNc #1#2#3#4
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN #3
    \cs:w #4 \cs_end:
  }
\cs_new:Npn \exp_args:NcNo #1#2#3#4
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN #3
    \exp_after:wN {#4}
  }
\cs_new:Npn \exp_args:Ncco #1#2#3#4
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \cs:w #3 \exp_after:wN \cs_end:
    \exp_after:wN {#4}
  }
```

(*End of definition for* `\exp_args:NNNV` *and others. These functions are documented on page* 38.)

`\exp_args:Nx`

```
\cs_new_protected:Npn \exp_args:Nx #1#2
  { \use:x { \exp_not:N #1 {#2} } }
```

(*End of definition for* `\exp_args:Nx`. *This function is documented on page 37.*)

43.3 Last-unbraced versions

`\__exp_arg_last_unbraced:nn`
`\::o_unbraced`
`\::V_unbraced`
`\::v_unbraced`
`\::e_unbraced`
`\::f_unbraced`
`\::x_unbraced`

There are a few places where the last argument needs to be available unbraced. First some helper macros.

```
\cs_new:Npn \__exp_arg_last_unbraced:nn #1#2 { #2#1 }
\cs_new:Npn \::o_unbraced \::: #1#2
  { \exp_after:wN \__exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
\cs_new:Npn \::V_unbraced \::: #1#2
  {
    \exp_after:wN \__exp_arg_last_unbraced:nn
      \exp_after:wN { \exp:w \__exp_eval_register:N #2 } {#1}
  }
\cs_new:Npn \::v_unbraced \::: #1#2
  {
    \exp_after:wN \__exp_arg_last_unbraced:nn
      \exp_after:wN { \exp:w \__exp_eval_register:c {#2} } {#1}
  }
\cs_new:Npn \::e_unbraced \::: #1#2
  { \tex_expanded:D { \exp_not:n {#1} #2 } }
\cs_new:Npn \::f_unbraced \::: #1#2
  {
    \exp_after:wN \__exp_arg_last_unbraced:nn
      \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } {#1}
  }
\cs_new_protected:Npn \::x_unbraced \::: #1#2
  {
    \cs_set_nopar:Npe \l__exp_internal_tl { \exp_not:n {#1} #2 }
    \l__exp_internal_tl
  }
```

(*End of definition for* `\__exp_arg_last_unbraced:nn` *and others. These functions are documented on page 44.*)

`\exp_last_unbraced:No`
`\exp_last_unbraced:NV`
`\exp_last_unbraced:Nv`
`\exp_last_unbraced:Ne`
`\exp_last_unbraced:Nf`
`\exp_last_unbraced:NNo`
`\exp_last_unbraced:NNV`
`\exp_last_unbraced:NNf`
`\exp_last_unbraced:Nco`
`\exp_last_unbraced:NcV`
`\exp_last_unbraced:NNNo`
`\exp_last_unbraced:NNNV`
`\exp_last_unbraced:NNNf`
`\exp_last_unbraced:Nno`
`\exp_last_unbraced:Nnf`
`\exp_last_unbraced:Noo`
`\exp_last_unbraced:Nfo`
`\exp_last_unbraced:NnNo`
`\exp_last_unbraced:NNNNo`
`\exp_last_unbraced:NNNNf`
`\exp_last_unbraced:Nx`

Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```
\cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
\cs_new:Npn \exp_last_unbraced:NV #1#2
  { \exp_after:wN #1 \exp:w \__exp_eval_register:N #2 }
\cs_new:Npn \exp_last_unbraced:Nv #1#2
  { \exp_after:wN #1 \exp:w \__exp_eval_register:c {#2} }
\cs_new:Npn \exp_last_unbraced:Ne #1#2
  { \exp_after:wN #1 \tex_expanded:D {#2} }
\cs_new:Npn \exp_last_unbraced:Nf #1#2
  { \exp_after:wN #1 \exp:w \exp_end_continue_f:w #2 }
\cs_new:Npn \exp_last_unbraced:NNo #1#2#3
  { \exp_after:wN #1 \exp_after:wN #2 #3 }
\cs_new:Npn \exp_last_unbraced:NNV #1#2#3
  {
```

```
    \exp_after:wN #1
    \exp_after:wN #2
    \exp:w \__exp_eval_register:N #3
  }
\cs_new:Npn \exp_last_unbraced:NNf #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp:w \exp_end_continue_f:w #3
  }
\cs_new:Npn \exp_last_unbraced:Nco #1#2#3
  { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
\cs_new:Npn \exp_last_unbraced:NcV #1#2#3
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp:w \__exp_eval_register:N #3
  }
\cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
  { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
\cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \exp:w \__exp_eval_register:N #4
  }
\cs_new:Npn \exp_last_unbraced:NNNf #1#2#3#4
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \exp:w \exp_end_continue_f:w #4
  }
\cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \::: }
\cs_new:Npn \exp_last_unbraced:Nnf { \::n \::f_unbraced \::: }
\cs_new:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \::: }
\cs_new:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \::: }
\cs_new:Npn \exp_last_unbraced:NnNo { \::n \::N \::o_unbraced \::: }
\cs_new:Npn \exp_last_unbraced:NNNNo #1#2#3#4#5
  { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 \exp_after:wN #4 #5 }
\cs_new:Npn \exp_last_unbraced:NNNNf #1#2#3#4#5
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN #4
    \exp:w \exp_end_continue_f:w #5
  }
\cs_new_protected:Npn \exp_last_unbraced:Nx { \::x_unbraced \::: }
```

(*End of definition for* `\exp_last_unbraced:No` *and others. These functions are documented on page 40.*)

`\exp_last_two_unbraced:Noo`
`\__exp_last_two_unbraced:noN`

If #2 is a single token then this can be implemented as

```
\cs_new:Npn \exp_last_two_unbraced:Noo #1 #2 #3
  { \exp_after:wN \exp_after:wN \exp_after:wN #1 \exp_after:wN #2 #3 }
```

However, for robustness this is not suitable. Instead, a bit of a shuffle is used to ensure that #2 can be multiple tokens.

```
\cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
  { \exp_after:wN \__exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
\cs_new:Npn \__exp_last_two_unbraced:noN #1#2#3
  { \exp_after:wN #3 #2 #1 }
```

(*End of definition for* `\exp_last_two_unbraced:Noo` *and* `\__exp_last_two_unbraced:noN`. *This function is documented on page 40.*)

43.4 Preventing expansion

`\__kernel_exp_not:w`

At the kernel level, we need the primitive behaviour to allow expansion *before* the brace group.

```
\cs_new_eq:NN \__kernel_exp_not:w \tex_unexpanded:D
```

(*End of definition for* `\__kernel_exp_not:w`.)

`\exp_not:c`
`\exp_not:o`
`\exp_not:e`
`\exp_not:f`
`\exp_not:V`
`\exp_not:v`

All these except `\exp_not:c` call the kernel-internal `\__kernel_exp_not:w` namely `\tex_unexpanded:D`.

```
\cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
\cs_new:Npn \exp_not:o #1 { \__kernel_exp_not:w \exp_after:wN {#1} }
\cs_new:Npn \exp_not:e #1
  { \__kernel_exp_not:w \tex_expanded:D { {#1} } }
\cs_new:Npn \exp_not:f #1
  { \__kernel_exp_not:w \exp_after:wN { \exp:w \exp_end_continue_f:w #1 } }
\cs_new:Npn \exp_not:V #1
  {
    \__kernel_exp_not:w \exp_after:wN
      { \exp:w \__exp_eval_register:N #1 }
  }
\cs_new:Npn \exp_not:v #1
  {
    \__kernel_exp_not:w \exp_after:wN
      { \exp:w \__exp_eval_register:c {#1} }
  }
```

(*End of definition for* `\exp_not:c` *and others. These functions are documented on page 41.*)

43.5 Controlled expansion

`\exp:w`
`\exp_end:`
`\exp_end_continue_f:w`
`\exp_end_continue_f:nw`

To trigger a sequence of "arbitrarily" many expansions we need a method to invoke TEX's expansion mechanism in such a way that (a) we are able to stop it in a controlled manner and (b) the result of what triggered the expansion in the first place is null, i.e., that we do not get any unwanted side effects. There aren't that many possibilities in TEX; in fact the one explained below might well be the only one (as normally the result of expansion is not null).

The trick here is to make use of the fact that `\tex_romannumeral:D` expands the tokens following it when looking for a number and that its expansion is null if that number turns out to be zero or negative. So we use that to start the expansion sequence: `\exp:w` is set equal to `\tex_romannumeral:D` in l3basics. To stop the expansion sequence in a controlled way all we need to provide is a constant integer zero as part of expanded tokens. As this is an integer constant it immediately stops `\tex_romannumeral:D`'s search for a number. Again, the definition of `\exp_end:` as the integer constant zero is in l3basics. (Note that according to our specification all tokens we expand initiated by `\exp:w` are supposed to be expandable (as well as their replacement text in the expansion) so we will not encounter a "number" that actually result in a roman numeral being generated. Or if we do then the programmer made a mistake.)

If on the other hand we want to stop the initial expansion sequence but continue with an f-type expansion we provide the alphabetic constant `` `^^@ `` that also represents `0` but this time TEX's syntax for a ⟨*number*⟩ continues searching for an optional space (and it continues expansion doing that) — see TEXbook page 269 for details.

```
\group_begin:
  \tex_catcode:D `\^^@ = 13
  \cs_new_protected:Npn \exp_end_continue_f:w { `^^@ }
```

If the above definition ever appears outside its proper context the active character `^^@` will be executed so we turn this into an error. The test for existence covers the (unlikely) case that some other code has already defined `^^@`: this is true for example for `xmltex.tex`.

```
  \if_cs_exist:N ^^@
  \else:
    \cs_new:Npn ^^@
      { \msg_expandable_error:nn { kernel } { bad-exp-end-f } }
  \fi:
```

The same but grabbing an argument to remove spaces and braces.

```
  \cs_new:Npn \exp_end_continue_f:nw #1 { `^^@ #1 }
\group_end:
```

(*End of definition for* `\exp:w` *and others. These functions are documented on page 43.*)

43.6 Defining function variants

```
⟨@@=cs⟩
```

`\s__cs_mark`
`\s__cs_stop`

Internal scan marks. No l3quark yet, so do things by hand.

```
\cs_new_eq:NN \s__cs_mark \scan_stop:
\cs_new_eq:NN \s__cs_stop \scan_stop:
```

(*End of definition for* `\s__cs_mark` *and* `\s__cs_stop`.)

`\q__cs_recursion_stop`

Internal recursion quarks. No l3quark yet, so do things by hand.

```
\cs_new:Npn \q__cs_recursion_stop { \q__cs_recursion_stop }
```

(*End of definition for* `\q__cs_recursion_stop`.)

`\__cs_use_none_delimit_by_s_stop:w`
`\__cs_use_i_delimit_by_s_stop:nw`
`__cs_use_none_delimit_by_q_recursion_stop:w`

Internal scan marks.

```
\cs_new:Npn \__cs_use_none_delimit_by_s_stop:w #1 \s__cs_stop { }
\cs_new:Npn \__cs_use_i_delimit_by_s_stop:nw #1 #2 \s__cs_stop {#1}
\cs_new:Npn \__cs_use_none_delimit_by_q_recursion_stop:w
  #1 \q__cs_recursion_stop { }
```

(*End of definition for* `\__cs_use_none_delimit_by_s_stop:w`*,* `\__cs_use_i_delimit_by_s_stop:nw`*, and* `\__cs_use_none_delimit_by_q_recursion_stop:w`*.*)

`\cs_generate_variant:Nn`
`\cs_generate_variant:cn`

#1 : Base form of a function; *e.g.*, `\tl_set:Nn`

#2 : One or more variant argument specifiers; e.g., `{Nx,c,cx}`

After making sure that the base form exists, test whether it is protected or not and define `\__cs_tmp:w` as either `\cs_new:Npe` or `\cs_new_protected:Npe`, which is then used to define all the variants (except those involving x-expansion, always protected). Split up the original base function only once, to grab its name and signature. Then we wish to iterate through the comma list of variant argument specifiers, which we first convert to a string: the reason is explained later.

```
\cs_new_protected:Npn \cs_generate_variant:Nn #1#2
  {
    \__cs_generate_variant:N #1
    \use:e
      {
        \__cs_generate_variant:nnNN
          \cs_split_function:N #1
          \exp_not:N #1
          \tl_to_str:n {#2} ,
            \exp_not:N \scan_stop: ,
            \exp_not:N \q__cs_recursion_stop
      }
  }
\cs_new_protected:Npn \cs_generate_variant:cn
  { \exp_args:Nc \cs_generate_variant:Nn }
```

(*End of definition for* `\cs_generate_variant:Nn`*. This function is documented on page* *34**.*)

`\__cs_generate_variant:N`
`\__cs_generate_variant:ww`
`\__cs_generate_variant:wwNw`

The goal here is to pick up protected parent functions. There are four cases: the parent function can be a primitive or a macro, and can be expandable or not. For non-expandable primitives, all variants should be protected; skipping the `\else:` branch is safe because non-expandable primitives cannot be TeX conditionals.

The other case where variants should be protected is when the parent function is a protected macro: then `protected` appears in the meaning before the first occurrence of `macro`. The `ww` auxiliary removes everything in the meaning string after the first `ma`. We use `ma` rather than the full `macro` because the meaning of the `\firstmark` primitive (and four others) can contain an arbitrary string after a leading `firstmark:`. Then, look for `pr` in the part we extracted: no need to look for anything longer: the only strings we can have are an empty string, `\long␣`, `\protected␣`, `\protected\long␣`, `\first`, `\top`, `\bot`, `\splittop`, or `\splitbot`, with `\` replaced by the appropriate escape character. If `pr` appears in the part before `ma`, the first `\s__cs_mark` is taken as an argument of the `wwNw` auxiliary, and `#3` is `\cs_new_protected:Npe`, otherwise it is `\cs_new:Npe`.

```
\cs_new_protected:Npe \__cs_generate_variant:N #1
  {
    \exp_not:N \exp_after:wN \exp_not:N \if_meaning:w
      \exp_not:N \exp_not:N #1 #1
      \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npe
    \exp_not:N \else:
      \exp_not:N \exp_after:wN \exp_not:N \__cs_generate_variant:ww
        \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma }
          \s__cs_mark
```

```
      \s__cs_mark \cs_new_protected:Npe
      \tl_to_str:n { pr }
      \s__cs_mark \cs_new:Npe
      \s__cs_stop
    \exp_not:N \fi:
  }
\exp_last_unbraced:NNNNo
  \cs_new_protected:Npn \__cs_generate_variant:ww
    #1 { \tl_to_str:n { ma } } #2 \s__cs_mark
    { \__cs_generate_variant:wwNw #1 }
\exp_last_unbraced:NNNNo
  \cs_new_protected:Npn \__cs_generate_variant:wwNw
    #1 { \tl_to_str:n { pr } } #2 \s__cs_mark #3 #4 \s__cs_stop
    { \cs_set_eq:NN \__cs_tmp:w #3 }
```

(*End of definition for __cs_generate_variant:N, __cs_generate_variant:ww, and __cs_generate_-variant:wwNw.*)

__cs_generate_variant:nnNN

#1 : Base name.
#2 : Base signature.
#3 : Boolean.
#4 : Base function.

If the boolean is \c_false_bool, the base function has no colon and we abort with an error; otherwise, set off a loop through the desired variant forms. The original function is retained as #4 for efficiency.

```
\cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
  {
    \if_meaning:w \c_false_bool #3
      \msg_error:nne { kernel } { missing-colon }
        { \token_to_str:c {#1} }
      \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
    \fi:
    \__cs_generate_variant:Nnnw #4 {#1}{#2}
  }
```

(*End of definition for __cs_generate_variant:nnNN.*)

__cs_generate_variant:Nnnw

#1 : Base function.
#2 : Base name.
#3 : Base signature.
#4 : Beginning of variant signature.

First check whether to terminate the loop over variant forms. Then, for each variant form, construct a new function name using the original base name, the variant signature consisting of l letters and the last $k-l$ letters of the base signature (of length k). For example, for a base function \prop_put:Nnn which needs a cV variant form, we want the new signature to be cVn.

There are further subtleties:

- In \cs_generate_variant:Nn \foo:nnTF {xxTF}, we must define \foo:xxTF using \exp_args:Nxx, rather than a hypothetical \exp_args:NxxTF. Thus, we wish to trim a common trailing part from the base signature and the variant signature.

- In \cs_generate_variant:Nn \foo:on {ox}, the function \foo:ox must be defined using \exp_args:Nnx, not \exp_args:Nox, to avoid double o expansion.

- Lastly, `\cs_generate_variant:Nn \foo:on {xn}` must trigger an error, because we do not have a means to replace o-expansion by x-expansion. More generally, we can only convert N to c, or convert n to V, v, o, e, f, or x.

All this boils down to a few rules. Only n and N-type arguments can be replaced by `\cs_generate_variant:Nn`. Other argument types are allowed to be passed unchanged from the base form to the variant: in the process they are changed to n except for N and p-type arguments. A common trailing part is ignored.

We compare the base and variant signatures one character at a time within e-expansion. The result is given to `\__cs_generate_variant:wwNN` (defined later) in the form ⟨*processed variant signature*⟩ `\s__cs_mark` ⟨*errors*⟩ `\s__cs_stop` ⟨*base function*⟩ ⟨*new function*⟩. If all went well, ⟨*errors*⟩ is empty; otherwise, it is a kernel error message and some clean-up code.

Note the space after #3 and after the following brace group. Those are ignored by TeX when fetching the last argument for `\__cs_generate_variant_loop:nNwN`, but can be used as a delimiter for `\__cs_generate_variant_loop_end:nwwwNNnn`.

```
\cs_new_protected:Npn \__cs_generate_variant:Nnnw #1#2#3#4 ,
  {
    \if_meaning:w \scan_stop: #4
      \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
    \fi:
    \use:e
      {
        \exp_not:N \__cs_generate_variant:wwNN
        \__cs_generate_variant_loop:nNwN { }
          #4
          \__cs_generate_variant_loop_end:nwwwNNnn
          \s__cs_mark
          #3 ~
          { ~ { } \fi: \__cs_generate_variant_loop_long:wNNnn } ~
          { }
          \s__cs_stop
        \exp_not:N #1 {#2} {#4}
      }
    \__cs_generate_variant:Nnnw #1 {#2} {#3}
  }
```

(*End of definition for* `\__cs_generate_variant:Nnnw`.)

`\__cs_generate_variant_loop:nNwN`
`\__cs_generate_variant_loop_base:N`
`\__cs_generate_variant_loop_same:w`
`\__cs_generate_variant_loop_end:nwwwNNnn`
`\__cs_generate_variant_loop_long:wNNnn`
`\__cs_generate_variant_loop_invalid:NNwNNnn`
`\__cs_generate_variant_loop_special:NNwNNnn`

#1 : Last few consecutive letters common between the base and variant (more precisely, `\__cs_generate_variant_same:N` ⟨*letter*⟩ for each letter).
#2 : Next variant letter.
#3 : Remainder of variant form.
#4 : Next base letter.

The first argument is populated by `\__cs_generate_variant_loop_same:w` when a variant letter and a base letter match. It is flushed into the input stream whenever the two letters are different: if the loop ends before, the argument is dropped, which means that trailing common letters are ignored.

The case where the two letters are different is only allowed if the base is N and the variant is c, or when the base is n and the variant is V, v, o, e, f, or x. Otherwise, call `\__cs_generate_variant_loop_invalid:NNwNNnn` to remove the end of the loop, get arguments at the end of the loop, and place an appropriate error message as a second argument of `\__cs_generate_variant:wwNN`. If the letters are distinct and the base letter is indeed `n` or `N`, leave in the input stream whatever argument `#1` was collected, and the next variant letter `#2`, then loop by calling `\__cs_generate_variant_loop:nNwN`.

The loop can stop in three ways.

- If the end of the variant form is encountered first, `#2` is `\__cs_generate_variant_loop_end:nwwwNNnn` (expanded by the conditional `\if:w`), which inserts some tokens to end the conditional; grabs the ⟨*base name*⟩ as `#7`, the ⟨*variant signature*⟩ `#8`, the ⟨*next base letter*⟩ `#1` and the part `#3` of the base signature that wasn't read yet; and combines those into the ⟨*new function*⟩ to be defined.
- If the end of the base form is encountered first, `#4` is `~{}\fi:` which ends the conditional (with an empty expansion), followed by `\__cs_generate_variant_loop_long:wNNnn`, which places an error as the second argument of `\__cs_generate_variant:wwNN`.
- The loop can be interrupted early if the requested expansion is unavailable, namely when the variant and base letters differ and the base is not the right one (`n` or `N` to support the variant). In that case too an error is placed as the second argument of `\__cs_generate_variant:wwNN`.

Note that if the variant form has the same length as the base form, `#2` is as described in the first point, and `#4` as described in the second point above. The `\__cs_generate_variant_loop_end:nwwwNNnn` breaking function takes the empty brace group in `#4` as its first argument: this empty brace group produces the correct signature for the full variant.

```
\cs_new:Npn \__cs_generate_variant_loop:nNwN #1#2#3 \s__cs_mark #4
  {
    \if:w #2 #4
      \exp_after:wN \__cs_generate_variant_loop_same:w
    \else:
      \if:w #4 \__cs_generate_variant_loop_base:N #2 \else:
        \if:w 0
          \if:w N #4 \else: \if:w n #4 \else: 1 \fi: \fi:
          \if:w \scan_stop: \__cs_generate_variant_loop_base:N #2 1 \fi:
          0
          \__cs_generate_variant_loop_special:NNwNNnn #4#2
        \else:
          \__cs_generate_variant_loop_invalid:NNwNNnn #4#2
        \fi:
      \fi:
    \fi:
    #1
    \prg_do_nothing:
    #2
    \__cs_generate_variant_loop:nNwN { } #3 \s__cs_mark
  }
\cs_new:Npn \__cs_generate_variant_loop_base:N #1
  {
    \if:w c #1 N \else:
      \if:w o #1 n \else:
        \if:w V #1 n \else:
          \if:w v #1 n \else:
```

```
            \if:w f #1 n \else:
              \if:w e #1 n \else:
                \if:w x #1 n \else:
                  \if:w n #1 n \else:
                    \if:w N #1 N \else:
                      \scan_stop:
                    \fi:
                  \fi:
                \fi:
              \fi:
            \fi:
          \fi:
        \fi:
      \fi:
    \fi:
  }
\cs_new:Npn \__cs_generate_variant_loop_same:w
    #1 \prg_do_nothing: #2#3#4
  { #3 { #1 \__cs_generate_variant_same:N #2 } }
\cs_new:Npn \__cs_generate_variant_loop_end:nwwwNNnn
    #1#2 \s__cs_mark #3 ~ #4 \s__cs_stop #5#6#7#8
  {
    \scan_stop: \scan_stop: \fi:
    \s__cs_mark \s__cs_stop
    \exp_not:N #6
    \exp_not:c { #7 : #8 #1 #3 }
  }
\cs_new:Npn \__cs_generate_variant_loop_long:wNNnn #1 \s__cs_stop #2#3#4#5
  {
    \exp_not:n
      {
        \s__cs_mark
        \msg_error:nnee { kernel } { variant-too-long }
          {#5} { \token_to_str:N #3 }
        \use_none:nnn
        \s__cs_stop
        #3
        #3
      }
  }
\cs_new:Npn \__cs_generate_variant_loop_invalid:NNwNNnn
    #1#2 \fi: \fi: \fi: #3 \s__cs_stop #4#5#6#7
  {
    \fi: \fi: \fi:
    \exp_not:n
      {
        \s__cs_mark
        \msg_error:nneeee { kernel } { invalid-variant }
          {#7} { \token_to_str:N #5 } {#1} {#2}
        \use_none:nnn
        \s__cs_stop
        #5
        #5
      }
```

```
  }
\cs_new:Npn \__cs_generate_variant_loop_special:NNwNNnn
  #1#2#3 \s__cs_stop #4#5#6#7
  {
    #3 \s__cs_stop #4 #5 {#6} {#7}
    \exp_not:n
      {
        \msg_error:nneeee
          { kernel } { deprecated-variant }
          {#7} { \token_to_str:N #5 } {#1} {#2}
      }
  }
```

(*End of definition for* `\__cs_generate_variant_loop:nNwN` *and others.*)

`\__cs_generate_variant_same:N`

When the base and variant letters are identical, don't do any expansion. For most argument types, we can use the `n`-type no-expansion, but the `N` and `p` types require a slightly different behaviour with respect to braces. For `V`-type this function could output `N` to avoid adding useless braces but that is not a problem.

```
\cs_new:Npn \__cs_generate_variant_same:N #1
  {
    \if:w N #1 #1 \else:
      \if:w p #1 #1 \else:
        \token_to_str:N n
        \if:w n #1 \else:
          \__cs_generate_variant_loop_special:NNwNNnn #1#1
        \fi:
      \fi:
    \fi:
  }
```

(*End of definition for* `\__cs_generate_variant_same:N`.)

`\__cs_generate_variant:wwNN`

If the variant form has already been defined, log its existence (provided `log-functions` is active). Otherwise, make sure that the `\exp_args:N #3` form is defined, and if it contains `x`, change `\__cs_tmp:w` locally to `\cs_new_protected:Npe`. Then define the variant by combining the `\exp_args:N #3` variant and the base function.

```
\cs_new_protected:Npn \__cs_generate_variant:wwNN
    #1 \s__cs_mark #2 \s__cs_stop #3#4
  {
    #2
    \cs_if_free:NT #4
      {
        \group_begin:
          \__cs_generate_internal_variant:n {#1}
          \__cs_tmp:w #4 { \exp_not:c { exp_args:N #1 } \exp_not:N #3 }
        \group_end:
      }
  }
```

(*End of definition for* `\__cs_generate_variant:wwNN`.)

`\__cs_generate_internal_variant:n`
`\__cs_generate_internal_variant_loop:n`

First test for the presence of `x` (this is where working with strings makes our lives easier), as the result should be protected, and the next variant to be defined using that internal variant should be protected (done by setting `\__cs_tmp:w`). Then call `\__cs_generate_internal_variant:NNn` with arguments `\cs_new_protected:cpn` `\use:x` (for protected) or `\cs_new:cpn` `\tex_expanded:D` (expandable) and the signature. If `p` appears in the signature, or if the function to be defined is expandable and the primitive `\expanded` is not available, or if there are more than 8 arguments, call some fall-back code that just puts the appropriate `\::` commands. Otherwise, call `\__cs_generate_internal_one_go:NNn` to construct the `\exp_args:N...` function as a macro taking up to 9 arguments and expanding them using `\use:x` or `\tex_expanded:D`.

```
\cs_new_protected:Npe \__cs_generate_internal_variant:n #1
  {
    \exp_not:N \__cs_generate_internal_variant:wwnNwn
      #1 \s__cs_mark
        { \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npe }
        \cs_new_protected:cpn
        \use:x
      \token_to_str:N x \s__cs_mark
        { }
        \cs_new:cpn
        \exp_not:N \tex_expanded:D
    \s__cs_stop
      {#1}
  }
\exp_last_unbraced:NNNNo
  \cs_new_protected:Npn \__cs_generate_internal_variant:wwnNwn #1
    { \token_to_str:N x } #2 \s__cs_mark #3#4#5#6 \s__cs_stop #7
  {
    #3
    \cs_if_free:cT { exp_args:N #7 }
      { \__cs_generate_internal_variant:NNn #4 #5 {#7} }
  }
\cs_set_protected:Npn \__cs_tmp:w #1
  {
    \cs_new_protected:Npn \__cs_generate_internal_variant:NNn ##1##2##3
      {
        \if_catcode:w X \use_none:nnnnnnnn ##3
            \prg_do_nothing: \prg_do_nothing: \prg_do_nothing:
            \prg_do_nothing: \prg_do_nothing: \prg_do_nothing:
            \prg_do_nothing: \prg_do_nothing: X
          \exp_after:wN \__cs_generate_internal_test:Nw \exp_after:wN ##2
        \else:
          \exp_after:wN \__cs_generate_internal_test_aux:w \exp_after:wN #1
        \fi:
        ##3
        \s__cs_mark
        {
          \use:e
            {
              ##1 { exp_args:N ##3 }
                { \__cs_generate_internal_variant_loop:n ##3 { : \use_i:nn } }
            }
        }
        #1
        \s__cs_mark
        { \exp_not:n { \__cs_generate_internal_one_go:NNn ##1 ##2 {##3} } }
```

```
        \s__cs_stop
      }
    \cs_new_protected:Npn \__cs_generate_internal_test_aux:w
        ##1 #1 ##2 \s__cs_mark ##3 ##4 \s__cs_stop {##3}
    \cs_new_eq:NN \__cs_generate_internal_test:Nw
      \__cs_generate_internal_test_aux:w
  }
\exp_args:No \__cs_tmp:w { \token_to_str:N p }
\cs_new_protected:Npn \__cs_generate_internal_one_go:NNn #1#2#3
  {
    \__cs_generate_internal_loop:nwnnw
      { \exp_not:N ##1 } 1 . { } { }
      #3 { ? \__cs_generate_internal_end:w } X ;
      23456789 { ? \__cs_generate_internal_long:w } ;
    #1 #2 {#3}
  }
\cs_new_protected:Npn \__cs_generate_internal_loop:nwnnw #1#2 . #3#4#5#6 ; #7
  {
    \use_none:n #5
    \use_none:n #7
    \cs_if_exist_use:cF { __cs_generate_internal_#5:NN }
      { \__cs_generate_internal_other:NN }
        #5 #7
    #7 .
    { #3 #1 } { #4 ## #2 }
    #6 ;
  }
\cs_new_protected:Npn \__cs_generate_internal_N:NN #1#2
  { \__cs_generate_internal_loop:nwnnw { \exp_not:N ###2 } }
\cs_new_protected:Npn \__cs_generate_internal_c:NN #1#2
  { \exp_args:No \__cs_generate_internal_loop:nwnnw { \exp_not:c {###2} } }
\cs_new_protected:Npn \__cs_generate_internal_n:NN #1#2
  { \__cs_generate_internal_loop:nwnnw { { \exp_not:n {###2} } } }
\cs_new_protected:Npn \__cs_generate_internal_x:NN #1#2
  { \__cs_generate_internal_loop:nwnnw { {###2} } }
\cs_new_protected:Npn \__cs_generate_internal_other:NN #1#2
  {
    \exp_args:No \__cs_generate_internal_loop:nwnnw
      {
        \exp_after:wN
        {
          \exp:w \exp_args:NNc \exp_after:wN \exp_end:
          { exp_not:#1 } {###2}
        }
      }
  }
\cs_new_protected:Npn \__cs_generate_internal_end:w #1 . #2#3#4 ; #5 ; #6#7#8
  { #6 { exp_args:N #8 } #3 { #7 {#2} } }
\cs_new_protected:Npn \__cs_generate_internal_long:w #1 N #2#3 . #4#5#6#
  {
    \exp_args:Nx \__cs_generate_internal_long:nnnNNn
      { \__cs_generate_internal_variant_loop:n #2 #6 { : \use_i:nn } }
      {#4} {#5}
  }
```

```
\cs_new:Npn \__cs_generate_internal_long:nnnNNn #1#2#3#4 ; ; #5#6#7
  { #5 { exp_args:N #7 } #3 { #6 { \exp_not:n {#1} {#2} } } }
```

This command grabs char by char outputting `\::#1` (not expanded further). We avoid tests by putting a trailing `:` `\use_i:nn`, which leaves `\cs_end:` and removes the looping macro. The colon is in fact also turned into `\:::` so that the required structure for `\exp_args:N...` commands is correctly terminated.

```
\cs_new:Npn \__cs_generate_internal_variant_loop:n #1
  {
    \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
    \__cs_generate_internal_variant_loop:n
  }
```

(*End of definition for* `\__cs_generate_internal_variant:n` *and* `\__cs_generate_internal_variant_-loop:n`.)

\prg_generate_conditional_variant:Nnn
__cs_generate_variant:nnNnn
__cs_generate_variant:w
__cs_generate_variant:n
__cs_generate_variant_p_form:nnn
__cs_generate_variant_T_form:nnn
__cs_generate_variant_F_form:nnn
__cs_generate_variant_TF_form:nnn

```
\cs_new_protected:Npn \prg_generate_conditional_variant:Nnn #1
  {
    \use:e
      {
        \__cs_generate_variant:nnNnn
          \cs_split_function:N #1
      }
  }
\cs_new_protected:Npn \__cs_generate_variant:nnNnn #1#2#3#4#5
  {
    \if_meaning:w \c_false_bool #3
      \msg_error:nne { kernel } { missing-colon }
        { \token_to_str:c {#1} }
      \__cs_use_i_delimit_by_s_stop:nw
    \fi:
    \exp_after:wN \__cs_generate_variant:w
    \tl_to_str:n {#5} , \scan_stop: , \q__cs_recursion_stop
    \__cs_use_none_delimit_by_s_stop:w \s__cs_mark {#1} {#2} {#4} \s__cs_stop
  }
\cs_new_protected:Npn \__cs_generate_variant:w
    #1 , #2 \s__cs_mark #3#4#5
  {
    \if_meaning:w \scan_stop: #1 \scan_stop:
      \if_meaning:w \q__cs_nil #1 \q__cs_nil
        \use_i:nnn
      \fi:
      \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
    \else:
      \cs_if_exist_use:cTF { __cs_generate_variant_#1_form:nnn }
        { {#3} {#4} {#5} }
        {
          \msg_error:nnee
            { kernel } { conditional-form-unknown }
            {#1} { \token_to_str:c { #3 : #4 } }
        }
    \fi:
    \__cs_generate_variant:w #2 \s__cs_mark {#3} {#4} {#5}
  }
```

```
\cs_new_protected:Npn \__cs_generate_variant_p_form:nnn #1#2
  { \cs_generate_variant:cn { #1 _p : #2 } }
\cs_new_protected:Npn \__cs_generate_variant_T_form:nnn #1#2
  { \cs_generate_variant:cn { #1 : #2 T } }
\cs_new_protected:Npn \__cs_generate_variant_F_form:nnn #1#2
  { \cs_generate_variant:cn { #1 : #2 F } }
\cs_new_protected:Npn \__cs_generate_variant_TF_form:nnn #1#2
  { \cs_generate_variant:cn { #1 : #2 TF } }
```

(*End of definition for* `\prg_generate_conditional_variant:Nnn` *and others. This function is documented on page 66.*)

`\exp_args_generate:n`

This function is not used in the kernel hence we can use functions that are defined in later modules. It also does not need to be fast so use inline mappings. For each requested variant we check that there are no characters besides `NnpcofVvx`, in particular that there are no spaces. Then we just call the internal function.

```
\cs_new_protected:Npn \exp_args_generate:n #1
  {
    \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
      {
        \str_map_inline:nn {##1}
          {
            \str_if_in:nnF { NnpcofeVvx } {####1}
              {
                \msg_error:nnnn { kernel } { invalid-exp-args }
                  {####1} {##1}
                \str_map_break:n { \use_none:nn }
              }
          }
        \__cs_generate_internal_variant:n {##1}
      }
  }
```

(*End of definition for* `\exp_args_generate:n`*. This function is documented on page 35.*)

43.7 Definitions with the automated technique

Some of these could be done more efficiently, but the complexity of coding then becomes an issue. Notice that the auto-generated functions actually take no arguments themselves.

`\exp_args:Nnc`
`\exp_args:Nno`
`\exp_args:NnV`
`\exp_args:Nnv`
`\exp_args:Nne`
`\exp_args:Nnf`
`\exp_args:Noc`
`\exp_args:Noo`
`\exp_args:Nof`
`\exp_args:NVo`
`\exp_args:Nfo`
`\exp_args:Nff`
`\exp_args:Nee`
`\exp_args:NNx`
`\exp_args:Ncx`
`\exp_args:Nnx`
`\exp_args:Nox`
`\exp_args:Nxo`
`\exp_args:Nxx`

Here are the actual function definitions, using the helper functions above. The group is used because `\__cs_generate_internal_variant:n` redefines `\__cs_tmp:w` locally.

```
\cs_set_protected:Npn \__cs_tmp:w #1
  {
    \group_begin:
      \exp_args:No \__cs_generate_internal_variant:n
        { \tl_to_str:n {#1} }
    \group_end:
  }
\__cs_tmp:w { nc }
\__cs_tmp:w { no }
\__cs_tmp:w { nV }
\__cs_tmp:w { nv }
```

```
\__cs_tmp:w { ne }
\__cs_tmp:w { nf }
\__cs_tmp:w { oc }
\__cs_tmp:w { oo }
\__cs_tmp:w { of }
\__cs_tmp:w { Vo }
\__cs_tmp:w { fo }
\__cs_tmp:w { ff }
\__cs_tmp:w { ee }
\__cs_tmp:w { Nx }
\__cs_tmp:w { cx }
\__cs_tmp:w { nx }
\__cs_tmp:w { ox }
\__cs_tmp:w { xo }
\__cs_tmp:w { xx }
```

(*End of definition for* `\exp_args:Nnc` *and others. These functions are documented on page 38.*)

\exp_args:NNcf
\exp_args:NNno
\exp_args:NNnV
\exp_args:NNoo
\exp_args:NNVV
\exp_args:Ncno
\exp_args:NcnV
\exp_args:Ncoo
\exp_args:NcVV
\exp_args:Nnnc
\exp_args:Nnno
\exp_args:Nnnf
\exp_args:Nnff
\exp_args:Nooo
\exp_args:Noof
\exp_args:Nffo
\exp_args:Neee
\exp_args:NNNx
\exp_args:NNnx
\exp_args:NNox
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Noox

```
\__cs_tmp:w { Ncf }
\__cs_tmp:w { Nno }
\__cs_tmp:w { NnV }
\__cs_tmp:w { Noo }
\__cs_tmp:w { NVV }
\__cs_tmp:w { cno }
\__cs_tmp:w { cnV }
\__cs_tmp:w { coo }
\__cs_tmp:w { cVV }
\__cs_tmp:w { nnc }
\__cs_tmp:w { nno }
\__cs_tmp:w { nnf }
\__cs_tmp:w { nff }
\__cs_tmp:w { ooo }
\__cs_tmp:w { oof }
\__cs_tmp:w { ffo }
\__cs_tmp:w { eee }
\__cs_tmp:w { NNx }
\__cs_tmp:w { Nnx }
\__cs_tmp:w { Nox }
\__cs_tmp:w { nnx }
\__cs_tmp:w { nox }
\__cs_tmp:w { ccx }
\__cs_tmp:w { cnx }
\__cs_tmp:w { oox }
```

(*End of definition for* `\exp_args:NNcf` *and others. These functions are documented on page 39.*)

43.8 Held-over variant generation

\cs_generate_from_arg_count:NNno
\cs_replacement_spec:c

A couple of variants that are from early functions.

```
\cs_generate_variant:Nn \cs_generate_from_arg_count:NNnn { NNno }
\cs_generate_variant:Nn \cs_replacement_spec:N { c }
```

(*End of definition for* `\cs_generate_from_arg_count:NNnn` *and* `\cs_replacement_spec:N`*. These functions are documented on page 20.*)

```
⟨/package⟩
```

Chapter 44

l3sort implementation

```
⟨*package⟩
⟨@@=sort⟩
```

44.1 Variables

\g__sort_internal_seq
\g__sort_internal_tl

Sorting happens in a group; the result is stored in those global variables before being copied outside the group to the proper places. For seq and tl this is more efficient than using `\use:e` (or some `\exp_args:NNNe`) to smuggle the definition outside the group since TeX does not need to re-read tokens. For clist we don't gain anything since the result is converted from seq to clist anyways.

```
\seq_new:N \g__sort_internal_seq
\tl_new:N \g__sort_internal_tl
```

(*End of definition for* `\g__sort_internal_seq` *and* `\g__sort_internal_tl`.)

\l__sort_length_int
\l__sort_min_int
\l__sort_top_int
\l__sort_max_int
\l__sort_true_max_int

The sequence has `\l__sort_length_int` items and is stored from `\l__sort_min_int` to `\l__sort_top_int` $-$ 1. While reading the sequence in memory, we check that `\l__sort_top_int` remains at most `\l__sort_max_int`, precomputed by `\__sort_compute_range:`. That bound is such that the merge sort only uses `\toks` registers less than `\l__sort_true_max_int`, namely those that have not been allocated for use in other code: the user's comparison code could alter these.

```
\int_new:N \l__sort_length_int
\int_new:N \l__sort_min_int
\int_new:N \l__sort_top_int
\int_new:N \l__sort_max_int
\int_new:N \l__sort_true_max_int
```

(*End of definition for* `\l__sort_length_int` *and others.*)

\l__sort_block_int

Merge sort is done in several passes. In each pass, blocks of size `\l__sort_block_int` are merged in pairs. The block size starts at 1, and, for a length in the range $[2^k+1, 2^{k+1}]$, reaches 2^k in the last pass.

```
\int_new:N \l__sort_block_int
```

(*End of definition for* `\l__sort_block_int`.)

`\l__sort_begin_int`
`\l__sort_end_int`

When merging two blocks, `\l__sort_begin_int` marks the lowest index in the two blocks, and `\l__sort_end_int` marks the highest index, plus 1.

```
\int_new:N \l__sort_begin_int
\int_new:N \l__sort_end_int
```

(*End of definition for* `\l__sort_begin_int` *and* `\l__sort_end_int`*.*)

`\l__sort_A_int`
`\l__sort_B_int`
`\l__sort_C_int`

When merging two blocks (whose end-points are `beg` and `end`), A starts from the high end of the low block, and decreases until reaching `beg`. The index B starts from the top of the range and marks the register in which a sorted item should be put. Finally, C points to the copy of the high block in the interval of registers starting at `\l__sort_length_int`, upwards. C starts from the upper limit of that range.

```
\int_new:N \l__sort_A_int
\int_new:N \l__sort_B_int
\int_new:N \l__sort_C_int
```

(*End of definition for* `\l__sort_A_int`*,* `\l__sort_B_int`*, and* `\l__sort_C_int`*.*)

`\s__sort_mark`
`\s__sort_stop`

Internal scan marks.

```
\scan_new:N \s__sort_mark
\scan_new:N \s__sort_stop
```

(*End of definition for* `\s__sort_mark` *and* `\s__sort_stop`*.*)

44.2 Finding available `\toks` registers

`\__sort_shrink_range:`
`\__sort_shrink_range_loop:`

After `\__sort_compute_range:` (defined below) determines that `\toks` registers between `\l__sort_min_int` (included) and `\l__sort_true_max_int` (excluded) have not yet been assigned, `\__sort_shrink_range:` computes `\l__sort_max_int` to reflect the need for a buffer when merging blocks in the merge sort. Given $2^n \leq A \leq 2^n + 2^{n-1}$ registers we can sort $\lfloor A/2 \rfloor + 2^{n-2}$ items while if we have $2^n + 2^{n-1} \leq A \leq 2^{n+1}$ registers we can sort $A - 2^{n-1}$ items. We first find out a power 2^n such that $2^n \leq A \leq 2^{n+1}$ by repeatedly halving `\l__sort_block_int`, starting at 2^{15} or 2^{14} namely half the total number of registers, then we use the formulas and set `\l__sort_max_int`.

```
\cs_new_protected:Npn \__sort_shrink_range:
  {
    \int_set:Nn \l__sort_A_int
      { \l__sort_true_max_int - \l__sort_min_int + 1 }
    \int_set:Nn \l__sort_block_int { \c_max_register_int / 2 }
    \__sort_shrink_range_loop:
    \int_set:Nn \l__sort_max_int
      {
        \int_compare:nNnTF
          { \l__sort_block_int * 3 / 2 } > \l__sort_A_int
          {
            \l__sort_min_int
            + ( \l__sort_A_int - 1 ) / 2
            + \l__sort_block_int / 4
            - 1
          }
          { \l__sort_true_max_int - \l__sort_block_int / 2 }
      }
```

```
  }
\cs_new_protected:Npn \__sort_shrink_range_loop:
  {
    \if_int_compare:w \l__sort_A_int < \l__sort_block_int
      \tex_divide:D \l__sort_block_int 2 \exp_stop_f:
      \exp_after:wN \__sort_shrink_range_loop:
    \fi:
  }
```

(*End of definition for* `\__sort_shrink_range:` *and* `\__sort_shrink_range_loop:`.)

`\__sort_compute_range:`
`\__sort_redefine_compute_range:`
`\c__sort_max_length_int`

First find out what `\toks` have not yet been assigned. There are many cases. In LaTeX 2_ε with no package, available `\toks` range from `\count`$15+1$ to `\c_max_register_int` included (this was not altered despite the 2015 changes). When `\loctoks` is defined, namely in plain (e)TeX, or when the package etex is loaded in LaTeX 2_ε, redefine `\__sort_compute_range:` to use the range `\count`265 to `\count`$275-1$. The elocalloc package also defines `\loctoks` but uses yet another number for the upper bound, namely `\e@alloc@top` (minus one). We must check for `\loctoks` every time a sorting function is called, as etex or elocalloc could be loaded.

In ConTeXt MkIV the range is from `\c_syst_last_allocated_toks`+1 to `\c_max_register_int`, and in MkII it is from `\lastallocatedtoks`+1 to `\c_max_register_int`. In all these cases, call `\__sort_shrink_range:`.

```
\cs_new_protected:Npn \__sort_compute_range:
  {
    \int_set:Nn \l__sort_min_int { \tex_count:D 15 + 1 }
    \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
    \__sort_shrink_range:
    \if_meaning:w \loctoks \tex_undefined:D \else:
      \if_meaning:w \loctoks \scan_stop: \else:
        \__sort_redefine_compute_range:
        \__sort_compute_range:
      \fi:
    \fi:
  }
\cs_new_protected:Npn \__sort_redefine_compute_range:
  {
    \cs_if_exist:cTF { ver@elocalloc.sty }
      {
        \cs_gset_protected:Npn \__sort_compute_range:
          {
            \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
            \int_set_eq:NN \l__sort_true_max_int \e@alloc@top
            \__sort_shrink_range:
          }
      }
      {
        \cs_gset_protected:Npn \__sort_compute_range:
          {
            \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
            \int_set:Nn \l__sort_true_max_int { \tex_count:D 275 }
            \__sort_shrink_range:
          }
      }
```

```
  }
\cs_if_exist:NT \loctoks { \__sort_redefine_compute_range: }
\tl_map_inline:nn { \lastallocatedtoks \c_syst_last_allocated_toks }
  {
    \cs_if_exist:NT #1
      {
        \cs_gset_protected:Npn \__sort_compute_range:
          {
            \int_set:Nn \l__sort_min_int { #1 + 1 }
            \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
            \__sort_shrink_range:
          }
      }
  }
```

(*End of definition for* `\__sort_compute_range:`*,* `\__sort_redefine_compute_range:`*, and* `\c__sort_max_length_int`*.*)

44.3 Protected user commands

`\__sort_main:NNNn`

Sorting happens in three steps. First store items in `\toks` registers ranging from `\l__sort_min_int` to `\l__sort_top_int` $-$ 1, while checking that the list is not too long. If we reach the maximum length, that's an error; exit the group. Secondly, sort the array of `\toks` registers, using the user-defined sorting function: `\__sort_level:` calls `\__sort_compare:nn` as needed. Finally, unpack the `\toks` registers (now sorted) into the target tl, or into `\g__sort_internal_seq` for seq and clist. This is done by `\__sort_seq:NNNNn` and `\__sort_tl:NNn`.

```
\cs_new_protected:Npn \__sort_main:NNNn #1#2#3#4
  {
    \__sort_disable_toksdef:
    \__sort_compute_range:
    \int_set_eq:NN \l__sort_top_int \l__sort_min_int
    #1 #3
      {
        \if_int_compare:w \l__sort_top_int = \l__sort_max_int
          \__sort_too_long_error:NNw #2 #3
        \fi:
        \tex_toks:D \l__sort_top_int {##1}
        \int_incr:N \l__sort_top_int
      }
    \int_set:Nn \l__sort_length_int
      { \l__sort_top_int - \l__sort_min_int }
    \cs_set:Npn \__sort_compare:nn ##1 ##2 {#4}
    \int_set:Nn \l__sort_block_int { 1 }
    \__sort_level:
  }
```

(*End of definition for* `\__sort_main:NNNn`*.*)

`\tl_sort:Nn`
`\tl_sort:cn`
`\tl_gsort:Nn`
`\tl_gsort:cn`
`\__sort_tl:NNn`
`\__sort_tl_toks:w`

Call the main sorting function then unpack `\toks` registers outside the group into the target token list. The unpacking is done by `\__sort_tl_toks:w`; registers are numbered from `\l__sort_min_int` to `\l__sort_top_int` $-$ 1. For expansion behaviour we need a couple of primitives. The \tl_gclear:N reduces memory usage. The \prg_break_-point: is used by __sort_main:NNNn when the list is too long.

```
\cs_new_protected:Npn \tl_sort:Nn { \__sort_tl:NNn \tl_set_eq:NN }
\cs_generate_variant:Nn \tl_sort:Nn { c }
\cs_new_protected:Npn \tl_gsort:Nn { \__sort_tl:NNn \tl_gset_eq:NN }
\cs_generate_variant:Nn \tl_gsort:Nn { c }
\cs_new_protected:Npn \__sort_tl:NNn #1#2#3
  {
    \group_begin:
      \__sort_main:NNNn \tl_map_inline:Nn \tl_map_break:n #2 {#3}
      \__kernel_tl_gset:Nx \g__sort_internal_tl
        { \__sort_tl_toks:w \l__sort_min_int ; }
    \group_end:
    #1 #2 \g__sort_internal_tl
    \tl_gclear:N \g__sort_internal_tl
    \prg_break_point:
  }
\cs_new:Npn \__sort_tl_toks:w #1 ;
  {
    \if_int_compare:w #1 < \l__sort_top_int
      { \tex_the:D \tex_toks:D #1 }
      \exp_after:wN \__sort_tl_toks:w
      \int_value:w \int_eval:n { #1 + 1 } \exp_after:wN ;
    \fi:
  }
```

(*End of definition for \tl_sort:Nn and others. These functions are documented on page 124.*)

\seq_sort:Nn
\seq_sort:cn
\seq_gsort:Nn
\seq_gsort:cn
\clist_sort:Nn
\clist_sort:cn
\clist_gsort:Nn
\clist_gsort:cn
__sort_seq:NNNNn

Use the same general framework for seq and clist. Apply the general sorting code, then unpack \toks into \g__sort_internal_seq. Outside the group copy or convert (for clist) the data to the target variable. The \seq_gclear:N reduces memory usage. The \prg_break_point: is used by __sort_main:NNNn when the list is too long.

```
\cs_new_protected:Npn \seq_sort:Nn
  { \__sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_set_eq:NN }
\cs_generate_variant:Nn \seq_sort:Nn { c }
\cs_new_protected:Npn \seq_gsort:Nn
  { \__sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_gset_eq:NN }
\cs_generate_variant:Nn \seq_gsort:Nn { c }
\cs_new_protected:Npn \clist_sort:Nn
  {
    \__sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n
      \clist_set_from_seq:NN
  }
\cs_generate_variant:Nn \clist_sort:Nn { c }
\cs_new_protected:Npn \clist_gsort:Nn
  {
    \__sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n
      \clist_gset_from_seq:NN
  }
\cs_generate_variant:Nn \clist_gsort:Nn { c }
\cs_new_protected:Npn \__sort_seq:NNNNn #1#2#3#4#5
  {
    \group_begin:
      \__sort_main:NNNn #1 #2 #4 {#5}
```

```
      \seq_gclear:N \g__sort_internal_seq
      \int_step_inline:nnn
        \l__sort_min_int { \l__sort_top_int - 1 }
        {
          \seq_gput_right:Ne \g__sort_internal_seq
            { \tex_the:D \tex_toks:D ##1 }
        }
    \group_end:
    #3 #4 \g__sort_internal_seq
    \seq_gclear:N \g__sort_internal_seq
    \prg_break_point:
  }
```

(*End of definition for* `\seq_sort:Nn` *and others. These functions are documented on page* *157*.)

44.4 Merge sort

`\__sort_level:` This function is called once blocks of size `\l__sort_block_int` (initially 1) are each sorted. If the whole list fits in one block, then we are done (this also takes care of the case of an empty list or a list with one item). Otherwise, go through pairs of blocks starting from 0, then double the block size, and repeat.

```
\cs_new_protected:Npn \__sort_level:
  {
    \if_int_compare:w \l__sort_block_int < \l__sort_length_int
      \l__sort_end_int \l__sort_min_int
      \__sort_merge_blocks:
      \tex_advance:D \l__sort_block_int \l__sort_block_int
      \exp_after:wN \__sort_level:
    \fi:
  }
```

(*End of definition for* `\__sort_level:`.)

`\__sort_merge_blocks:` This function is called to merge a pair of blocks, starting at the last value of `\l__sort_end_int` (end-point of the previous pair of blocks). If shifting by one block to the right we reach the end of the list, then this pass has ended: the end of the list is sorted already. Otherwise, store the result of that shift in A, which indexes the first block starting from the top end. Then locate the end-point (maximum) of the second block: shift `end` upwards by one more block, but keeping it $\leq$ `top`. Copy this upper block of `\toks` registers in registers above `length`, indexed by C: this is covered by `\__sort_copy_block:`. Once this is done we are ready to do the actual merger using `\__sort_merge_blocks_aux:`, after shifting A, B and C so that they point to the largest index in their respective ranges rather than pointing just beyond those ranges. Of course, once that pair of blocks is merged, move on to the next pair.

```
\cs_new_protected:Npn \__sort_merge_blocks:
  {
    \l__sort_begin_int \l__sort_end_int
    \tex_advance:D \l__sort_end_int \l__sort_block_int
    \if_int_compare:w \l__sort_end_int < \l__sort_top_int
      \l__sort_A_int \l__sort_end_int
      \tex_advance:D \l__sort_end_int \l__sort_block_int
      \if_int_compare:w \l__sort_end_int > \l__sort_top_int
```

```
          \l__sort_end_int \l__sort_top_int
        \fi:
        \l__sort_B_int \l__sort_A_int
        \l__sort_C_int \l__sort_top_int
        \__sort_copy_block:
        \int_decr:N \l__sort_A_int
        \int_decr:N \l__sort_B_int
        \int_decr:N \l__sort_C_int
        \exp_after:wN \__sort_merge_blocks_aux:
        \exp_after:wN \__sort_merge_blocks:
      \fi:
    }
```

(*End of definition for* `\__sort_merge_blocks:`.)

`\__sort_copy_block:` We wish to store a copy of the "upper" block of `\toks` registers, ranging between the initial value of `\l__sort_B_int` (included) and `\l__sort_end_int` (excluded) into a new range starting at the initial value of `\l__sort_C_int`, namely `\l__sort_top_int`.

```
\cs_new_protected:Npn \__sort_copy_block:
  {
    \tex_toks:D \l__sort_C_int \tex_toks:D \l__sort_B_int
    \int_incr:N \l__sort_C_int
    \int_incr:N \l__sort_B_int
    \if_int_compare:w \l__sort_B_int = \l__sort_end_int
      \use_i:nn
    \fi:
    \__sort_copy_block:
  }
```

(*End of definition for* `\__sort_copy_block:`.)

`\__sort_merge_blocks_aux:` At this stage, the first block starts at `\l__sort_begin_int`, and ends at `\l__sort_A_int`, and the second block starts at `\l__sort_top_int` and ends at `\l__sort_C_int`. The result of the merger is stored at positions indexed by `\l__sort_B_int`, which starts at `\l__sort_end_int` -1 and decreases down to `\l__sort_begin_int`, covering the full range of the two blocks. In other words, we are building the merger starting with the largest values. The comparison function is defined to return either `swapped` or `same`. Of course, this means the arguments need to be given in the order they appear originally in the list.

```
\cs_new_protected:Npn \__sort_merge_blocks_aux:
  {
    \exp_after:wN \__sort_compare:nn \exp_after:wN
      { \tex_the:D \tex_toks:D \exp_after:wN \l__sort_A_int \exp_after:wN }
      \exp_after:wN { \tex_the:D \tex_toks:D \l__sort_C_int }
    \prg_do_nothing:
    \__sort_return_mark:w
    \__sort_return_mark:w
    \s__sort_mark
    \__sort_return_none_error:
  }
```

(*End of definition for* `\__sort_merge_blocks_aux:`.)

\sort_return_same:
\sort_return_swapped:
__sort_return_mark:w
__sort_return_none_error:
__sort_return_two_error:

Each comparison should call `\sort_return_same:` or `\sort_return_swapped:` exactly once. If neither is called, `\__sort_return_none_error:` is called, since the `return_mark` removes tokens until `\s__sort_mark`. If one is called, the `return_mark` auxiliary removes everything except `\__sort_return_same:w` (or its `swapped` analogue) followed by `\__sort_return_none_error:`. Finally if two or more are called, `\__sort_return_two_error:` ends up before any `\__sort_return_mark:w`, so that it produces an error.

```
\cs_new_protected:Npn \sort_return_same:
    #1 \__sort_return_mark:w #2 \s__sort_mark
  {
    #1
    #2
    \__sort_return_two_error:
    \__sort_return_mark:w
    \s__sort_mark
    \__sort_return_same:w
  }
\cs_new_protected:Npn \sort_return_swapped:
    #1 \__sort_return_mark:w #2 \s__sort_mark
  {
    #1
    #2
    \__sort_return_two_error:
    \__sort_return_mark:w
    \s__sort_mark
    \__sort_return_swapped:w
  }
\cs_new_protected:Npn \__sort_return_mark:w #1 \s__sort_mark { }
\cs_new_protected:Npn \__sort_return_none_error:
  {
    \msg_error:nnee { sort } { return-none }
      { \tex_the:D \tex_toks:D \l__sort_A_int }
      { \tex_the:D \tex_toks:D \l__sort_C_int }
    \__sort_return_same:w \__sort_return_none_error:
  }
\cs_new_protected:Npn \__sort_return_two_error:
  {
    \msg_error:nnee { sort } { return-two }
      { \tex_the:D \tex_toks:D \l__sort_A_int }
      { \tex_the:D \tex_toks:D \l__sort_C_int }
  }
```

(*End of definition for* `\sort_return_same:` *and others. These functions are documented on page 46.*)

__sort_return_same:w

If the comparison function returns `same`, then the second argument fed to `\__sort_compare:nn` should remain to the right of the other one. Since we build the merger starting from the right, we copy that `\toks` register into the allotted range, then shift the pointers B and C, and go on to do one more step in the merger, unless the second block has been exhausted: then the remainder of the first block is already in the correct registers and we are done with merging those two blocks.

```
\cs_new_protected:Npn \__sort_return_same:w #1 \__sort_return_none_error:
  {
    \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
    \int_decr:N \l__sort_B_int
```

```
    \int_decr:N \l__sort_C_int
    \if_int_compare:w \l__sort_C_int < \l__sort_top_int
      \use_i:nn
    \fi:
    \__sort_merge_blocks_aux:
  }
```

(*End of definition for* `\__sort_return_same:w`.)

`\__sort_return_swapped:w` If the comparison function returns `swapped`, then the next item to add to the merger is the first argument, contents of the `\toks` register A. Then shift the pointers A and B to the left, and go for one more step for the merger, unless the left block was exhausted (A goes below the threshold). In that case, all remaining `\toks` registers in the second block, indexed by C, are copied to the merger by `\__sort_merge_blocks_end:`.

```
\cs_new_protected:Npn \__sort_return_swapped:w #1 \__sort_return_none_error:
  {
    \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_A_int
    \int_decr:N \l__sort_B_int
    \int_decr:N \l__sort_A_int
    \if_int_compare:w \l__sort_A_int < \l__sort_begin_int
      \__sort_merge_blocks_end: \use_i:nn
    \fi:
    \__sort_merge_blocks_aux:
  }
```

(*End of definition for* `\__sort_return_swapped:w`.)

`\__sort_merge_blocks_end:` This function's task is to copy the `\toks` registers in the block indexed by C to the merger indexed by B. The end can equally be detected by checking when B reaches the threshold `begin`, or when C reaches `top`.

```
\cs_new_protected:Npn \__sort_merge_blocks_end:
  {
    \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
    \int_decr:N \l__sort_B_int
    \int_decr:N \l__sort_C_int
    \if_int_compare:w \l__sort_B_int < \l__sort_begin_int
      \use_i:nn
    \fi:
    \__sort_merge_blocks_end:
  }
```

(*End of definition for* `\__sort_merge_blocks_end:`.)

44.5 Expandable sorting

Sorting expandably is very different from sorting and assigning to a variable. Since tokens cannot be stored, they must remain in the input stream, and be read through at every step. It is thus necessarily much slower (at best $O(n^2 \ln n)$) than non-expandable sorting functions ($O(n \ln n)$).

A prototypical version of expandable quicksort is as follows. If the argument has no item, return nothing, otherwise partition, using the first item as a pivot (argument `#4` of `\__sort:nnNnn`). The arguments of `\__sort:nnNnn` are 1. items less than `#4`, 2. items greater or equal to `#4`, 3. comparison, 4. pivot, 5. next item to test. If `#5` is the tail of the list, call \tl_sort:nN on #1 and on #2, placing #4 in between; \use:ff expands the parts to make \tl_sort:nN f-expandable. Otherwise, compare #4 and #5 using #3. If they are ordered, place #5 amongst the "greater" items, otherwise amongst the "lesser" items, and continue partitioning.

```
\cs_new:Npn \tl_sort:nN #1#2
  {
    \tl_if_blank:nF {#1}
      {
        \__sort:nnNnn { } { } #2
          #1 \q__sort_recursion_tail \q__sort_recursion_stop
      }
  }
\cs_new:Npn \__sort:nnNnn #1#2#3#4#5
  {
    \quark_if_recursion_tail_stop_do:nn {#5}
      { \use:ff { \tl_sort:nN {#1} #3 {#4} } { \tl_sort:nN {#2} #3 } }
    #3 {#4} {#5}
      { \__sort:nnNnn {#1} { #2 {#5} } #3 {#4} }
      { \__sort:nnNnn { #1 {#5} } {#2} #3 {#4} }
  }
\cs_generate_variant:Nn \use:nn { ff }
```

There are quite a few optimizations available here: the code below is less legible, but more than twice as fast.

In the simple version of the code, __sort:nnNnn is called $O(n \ln n)$ times on average (the number of comparisons required by the quicksort algorithm). Hence most of our focus is on optimizing that function.

The first speed up is to avoid testing for the end of the list at every call to __sort:nnNnn. For this, the list is prepared by changing each ⟨*item*⟩ of the original token list into ⟨*command*⟩ {⟨*item*⟩}, just like sequences are stored. We arrange things such that the ⟨*command*⟩ is the ⟨*conditional*⟩ provided by the user: the loop over the ⟨*prepared tokens*⟩ then looks like

```
\cs_new:Npn \__sort_loop:wNn ... #6#7
{
#6 {⟨pivot⟩} {#7} ⟨loop big⟩ ⟨loop small⟩
⟨extra arguments⟩
}
\__sort_loop:wNn ... ⟨prepared tokens⟩
⟨end-loop⟩ {} \s__sort_stop
```

In this example, which matches the structure of __sort_quick_split_i:NnnnnNn and a few other functions below, the __sort_loop:wNn auxiliary normally receives the user's ⟨*conditional*⟩ as #6 and an ⟨*item*⟩ as #7. This is compared to the ⟨*pivot*⟩ (the argument #5, not shown here), and the ⟨*conditional*⟩ leaves the ⟨*loop big*⟩ or ⟨*loop small*⟩ auxiliary, which both have the same form as __sort_loop:wNn, receiving the next pair ⟨*conditional*⟩ {⟨*item*⟩} as #6 and #7. At the end, #6 is the ⟨*end-loop*⟩ function, which terminates the loop.

The second speed up is to minimize the duplicated tokens between the true and false branches of the conditional. For this, we introduce two versions of __sort:nnNnn, which receive the new item as `#1` and place it either into the list `#2` of items less than the pivot `#4` or into the list `#3` of items greater or equal to the pivot.

```
\cs_new:Npn \__sort_i:nnnnNn #1#2#3#4#5#6
  {
    #5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
      {#6} { #2 {#1} } {#3} {#4}
  }
\cs_new:Npn \__sort_ii:nnnnNn #1#2#3#4#5#6
  {
    #5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
      {#6} {#2} { #3 {#1} } {#4}
  }
```

Note that the two functions have the form of `\__sort_loop:wNn` above, receiving as `#5` the conditional or a function to end the loop. In fact, the lists `#2` and `#3` must be made of pairs ⟨*conditional*⟩ `{`⟨*item*⟩`}`, so we have to replace `{#6}` above by `{ #5 {#6} }`, and `{#1}` by `#1`. The actual functions have one more argument, so all argument numbers are shifted compared to this code.

The third speed up is to avoid `\use:ff` using a continuation-passing style: `\__sort_quick_split:NnNn` expects a list followed by `\s__sort_mark {`⟨*code*⟩`}`, and expands to ⟨*code*⟩ ⟨*sorted list*⟩. Sorting the two parts of the list around the pivot is done with

```
\__sort_quick_split:NnNn #2 ...  \s__sort_mark
{
\__sort_quick_split:NnNn #1 ...  \s__sort_mark {⟨code⟩}
{⟨pivot⟩}
}
```

Items which are larger than the ⟨*pivot*⟩ are sorted, then placed after code that sorts the smaller items, and after the (braced) ⟨*pivot*⟩.

The fourth speed up is avoid the recursive call to `\tl_sort:nN` with an empty first argument. For this, we introduce functions similar to the `\__sort_i:nnnnNn` of the last example, but aware of whether the list of ⟨*conditional*⟩ `{`⟨*item*⟩`}` read so far that are less than the pivot, and the list of those greater or equal, are empty or not: see `\__sort_quick_split:NnNn` and functions defined below. Knowing whether the lists are empty or not is useless if we do not use distinct ending codes as appropriate. The splitting auxiliaries communicate to the ⟨*end-loop*⟩ function (that is initially placed after the "prepared" list) by placing a specific ending function, ignored when looping, but useful at the end. In fact, the ⟨*end-loop*⟩ function does nothing but place the appropriate ending function in front of all its arguments. The ending functions take care of sorting non-empty sublists, placing the pivot in between, and the continuation before.

The final change in fact slows down the code a little, but is required to avoid memory issues: schematically, when TeX encounters

```
\use:n { \use:n { \use:n { ... } ... } ... }
```

the argument of the first `\use:n` is not completely read by the second `\use:n`, hence must remain in memory; then the argument of the second `\use:n` is not completely read when grabbing the argument of the third `\use:n`, hence must remain in memory, and so on. The memory consumption grows quadratically with the number of nested `\use:n`. In practice, this means that we must read everything until a trailing `\s__sort_stop` once in a while, otherwise sorting lists of more than a few thousand items would exhaust a typical TeX's memory.

`\tl_sort:nN` `\__sort_quick_prepare:Nnnn` `\__sort_quick_prepare_end:NNNnw` `\__sort_quick_cleanup:w`

The code within the `\exp_not:f` sorts the list, leaving in most cases a leading `\exp_not:f`, which stops the expansion, letting the result be return within `\exp_not:n`. We filter out the case of a list with no item, which would otherwise cause problems. Then prepare the token list `#1` by inserting the conditional `#2` before each item. The `prepare` auxiliary receives the conditional as `#1`, the prepared token list so far as `#2`, the next prepared item as `#3`, and the item after that as `#4`. The loop ends when `#4` contains `\prg_break_point:`, then the `prepare_end` auxiliary finds the prepared token list as `#4`. The scene is then set up for `\__sort_quick_split:NnNn`, which sorts the prepared list and perform the post action placed after `\s__sort_mark`, namely removing the trailing `\s__sort_stop` and `\s__sort_stop` and leaving `\exp_stop_f:` to stop f-expansion.

```
\cs_new:Npn \tl_sort:nN #1#2
  {
    \exp_not:f
      {
        \tl_if_blank:nF {#1}
          {
            \__sort_quick_prepare:Nnnn #2 { } { }
              #1
              { \prg_break_point: \__sort_quick_prepare_end:NNNnw }
            \s__sort_stop
          }
      }
  }
\cs_new:Npn \__sort_quick_prepare:Nnnn #1#2#3#4
  {
    \prg_break: #4 \prg_break_point:
    \__sort_quick_prepare:Nnnn #1 { #2 #3 } { #1 {#4} }
  }
\cs_new:Npn \__sort_quick_prepare_end:NNNnw #1#2#3#4#5 \s__sort_stop
  {
    \__sort_quick_split:NnNn #4 \__sort_quick_end:nnTFNn { }
    \s__sort_mark { \__sort_quick_cleanup:w \exp_stop_f: }
    \s__sort_mark \s__sort_stop
  }
\cs_new:Npn \__sort_quick_cleanup:w #1 \s__sort_mark \s__sort_stop {#1}
```

(*End of definition for* `\tl_sort:nN` *and others. This function is documented on page* *124*.)

`\__sort_quick_split:NnNn` `\__sort_quick_only_i:NnnnnNn` `\__sort_quick_only_ii:NnnnnNn` `\__sort_quick_split_i:NnnnnNn` `\__sort_quick_split_ii:NnnnnNn`

The `only_i`, `only_ii`, `split_i` and `split_ii` auxiliaries receive a useless first argument, the new item `#2` (that they append to either one of the next two arguments), the list `#3` of items less than the pivot, bigger items `#4`, the pivot `#5`, a ⟨*function*⟩ `#6`, and an item `#7`. The ⟨*function*⟩ is the user's ⟨*conditional*⟩ except at the end of the list where it is `\__sort_quick_end:nnTFNn`. The comparison is applied to the ⟨*pivot*⟩ and the ⟨*item*⟩, and calls the `only_i` or `split_i` auxiliaries if the ⟨*item*⟩ is smaller, and the `only_ii` or `split_ii` auxiliaries otherwise. In both cases, the next auxiliary goes to work right away, with no intermediate expansion that would slow down operations. Note that the argument `#2` left for the next call has the form ⟨*conditional*⟩ `{`⟨*item*⟩`}`, so that the lists `#3` and `#4` keep the right form to be fed to the next sorting function. The `split` auxiliary differs from these in that it is missing three of the arguments, which would be empty, and its first argument is always the user's ⟨*conditional*⟩ rather than an ending function.

```
\cs_new:Npn \__sort_quick_split:NnNn #1#2#3#4
  {
    #3 {#2} {#4} \__sort_quick_only_ii:NnnnnNn
      \__sort_quick_only_i:NnnnnNn
      \__sort_quick_single_end:nnnwnw
      { #3 {#4} } { } { } {#2}
  }
\cs_new:Npn \__sort_quick_only_i:NnnnnNn #1#2#3#4#5#6#7
  {
    #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
      \__sort_quick_only_i:NnnnnNn
      \__sort_quick_only_i_end:nnnwnw
      { #6 {#7} } { #3 #2 } { } {#5}
  }
\cs_new:Npn \__sort_quick_only_ii:NnnnnNn #1#2#3#4#5#6#7
  {
    #6 {#5} {#7} \__sort_quick_only_ii:NnnnnNn
      \__sort_quick_split_i:NnnnnNn
      \__sort_quick_only_ii_end:nnnwnw
      { #6 {#7} } { } { #4 #2 } {#5}
  }
\cs_new:Npn \__sort_quick_split_i:NnnnnNn #1#2#3#4#5#6#7
  {
    #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
      \__sort_quick_split_i:NnnnnNn
      \__sort_quick_split_end:nnnwnw
      { #6 {#7} } { #3 #2 } {#4} {#5}
  }
\cs_new:Npn \__sort_quick_split_ii:NnnnnNn #1#2#3#4#5#6#7
  {
    #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
      \__sort_quick_split_i:NnnnnNn
      \__sort_quick_split_end:nnnwnw
      { #6 {#7} } {#3} { #4 #2 } {#5}
  }
```

(*End of definition for* `\__sort_quick_split:NnNn` *and others.*)

`\__sort_quick_end:nnTFNn`
`\__sort_quick_single_end:nnnwnw`
`\__sort_quick_only_i_end:nnnwnw`
`\__sort_quick_only_ii_end:nnnwnw`
`\__sort_quick_split_end:nnnwnw`

The `\__sort_quick_end:nnTFNn` appears instead of the user's conditional, and receives as its arguments the pivot `#1`, a fake item `#2`, a `true` and a `false` branches `#3` and `#4`, followed by an ending function `#5` (one of the four auxiliaries here) and another copy `#6` of the fake item. All those are discarded except the function `#5`. This function receives lists `#1` and `#2` of items less than or greater than the pivot `#3`, then a continuation code `#5` just after `\s__sort_mark`. To avoid a memory problem described earlier, all of the ending functions read `#6` until `\s__sort_stop` and place `#6` back into the input stream. When the lists `#1` and `#2` are empty, the `single` auxiliary simply places the continuation `#5` before the pivot `{#3}`. When `#2` is empty, `#1` is sorted and placed before the pivot `{#3}`, taking care to feed the continuation `#5` as a continuation for the function sorting `#1`. When `#1` is empty, `#2` is sorted, and the continuation argument is used to place the continuation `#5` and the pivot `{#3}` before the sorted result. Finally, when both lists are non-empty, items larger than the pivot are sorted, then items less than the pivot, and the continuations are done in such a way to place the pivot in between.

```
\cs_new:Npn \__sort_quick_end:nnTFNn #1#2#3#4#5#6 {#5}
\cs_new:Npn \__sort_quick_single_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
  { #5 {#3} #6 \s__sort_stop }
\cs_new:Npn \__sort_quick_only_i_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
  {
    \__sort_quick_split:NnNn #1
      \__sort_quick_end:nnTFNn { } \s__sort_mark {#5}
    {#3}
    #6 \s__sort_stop
  }
\cs_new:Npn \__sort_quick_only_ii_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
  {
    \__sort_quick_split:NnNn #2
      \__sort_quick_end:nnTFNn { } \s__sort_mark { #5 {#3} }
    #6 \s__sort_stop
  }
\cs_new:Npn \__sort_quick_split_end:nnnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
  {
    \__sort_quick_split:NnNn #2 \__sort_quick_end:nnTFNn { } \s__sort_mark
      {
        \__sort_quick_split:NnNn #1
          \__sort_quick_end:nnTFNn { } \s__sort_mark {#5}
        {#3}
      }
    #6 \s__sort_stop
  }
```

(*End of definition for* `\__sort_quick_end:nnTFNn` *and others.*)

44.6 Messages

`\__sort_error:` Bailing out of the sorting code is a bit tricky. It may not be safe to use a delimited argument, so instead we redefine many l3sort commands to be trivial, with `\__sort_level:` jumping to the break point. This error recovery won't work in a group.

```
\cs_new_protected:Npn \__sort_error:
  {
    \cs_set_eq:NN \__sort_merge_blocks_aux: \prg_do_nothing:
    \cs_set_eq:NN \__sort_merge_blocks: \prg_do_nothing:
    \cs_set_protected:Npn \__sort_level: { \group_end: \prg_break: }
  }
```

(*End of definition for* `\__sort_error:`*.*)

`\__sort_disable_toksdef:`
`\__sort_disabled_toksdef:n`

While sorting, `\toksdef` is locally disabled to prevent users from using `\newtoks` or similar commands in their comparison code: the `\toks` registers that would be assigned are in use by l3sort. In format mode, none of this is needed since there is no `\toks` allocator.

```
\cs_new_protected:Npn \__sort_disable_toksdef:
  { \cs_set_eq:NN \toksdef \__sort_disabled_toksdef:n }
\cs_new_protected:Npn \__sort_disabled_toksdef:n #1
  {
```

```
    \msg_error:nne { sort } { toksdef }
      { \token_to_str:N #1 }
    \__sort_error:
    \tex_toksdef:D #1
  }
\msg_new:nnnn { sort } { toksdef }
  { Allocation~of~\iow_char:N\\toks~registers~impossible~while~sorting. }
  {
    The~comparison~code~used~for~sorting~a~list~has~attempted~to~
    define~#1~as~a~new~\iow_char:N\\toks~register~using~
    \iow_char:N\\newtoks~
    or~a~similar~command.~The~list~will~not~be~sorted.
  }
```

(*End of definition for* `\__sort_disable_toksdef:` *and* `\__sort_disabled_toksdef:n`.)

`\__sort_too_long_error:NNw` When there are too many items in a sequence, this is an error, and we clean up properly the mapping over items in the list: break using the type-specific breaking function `#1`.

```
\cs_new_protected:Npn \__sort_too_long_error:NNw #1#2 \fi:
  {
    \fi:
    \msg_error:nneee { sort } { too-large }
      { \token_to_str:N #2 }
      { \int_eval:n { \l__sort_true_max_int - \l__sort_min_int } }
      { \int_eval:n { \l__sort_top_int - \l__sort_min_int } }
    #1 \__sort_error:
  }
\msg_new:nnnn { sort } { too-large }
  { The~list~#1~is~too~long~to~be~sorted~by~TeX. }
  {
    TeX~has~#2~toks~registers~still~available:~
    this~only~allows~to~sort~with~up~to~#3~
    items.~The~list~will~not~be~sorted.
  }
```

(*End of definition for* `\__sort_too_long_error:NNw`.)

```
\msg_new:nnnn { sort } { return-none }
  { The~comparison~code~did~not~return. }
  {
    When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~
    did~not~call~
    \iow_char:N\\sort_return_same: ~nor~
    \iow_char:N\\sort_return_swapped: .~
    Exactly~one~of~these~should~be~called.
  }
\msg_new:nnnn { sort } { return-two }
  { The~comparison~code~returned~multiple~times. }
  {
    When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~called~
    \iow_char:N\\sort_return_same: ~or~
    \iow_char:N\\sort_return_swapped: ~multiple~times.~
    Exactly~one~of~these~should~be~called.
  }
\prop_gput:Nnn \g_msg_module_name_prop { sort } { LaTeX }
\prop_gput:Nnn \g_msg_module_type_prop { sort } { }
```

⟨/package⟩

Chapter 45

l3tl-analysis implementation

```
⟨@@=tl⟩
```

45.1 Internal functions

\s__tl The format used to store token lists internally uses the scan mark `\s__tl` as a delimiter.

(*End of definition for* `\s__tl`.)

45.2 Internal format

The task of the l3tl-analysis module is to convert token lists to an internal format which allows us to extract all the relevant information about individual tokens (category code, character code), as well as reconstruct the token list quickly. This internal format is used in l3regex where we need to support arbitrary tokens, and it is used in conversion functions in l3str-convert, where we wish to support clusters of characters instead of single tokens.

We thus need a way to encode any ⟨*token*⟩ (even begin-group and end-group character tokens) in a way amenable to manipulating tokens individually. The best we can do is to find ⟨*tokens*⟩ which both o-expand and e/x-expand to the given ⟨*token*⟩. Collecting more information about the category code and character code is also useful for regular expressions, since most regexes are catcode-agnostic. The internal format thus takes the form of a succession of items of the form

⟨*tokens*⟩ `\s__tl` ⟨*catcode*⟩ ⟨*char code*⟩ `\s__tl`

The ⟨*tokens*⟩ o- *and* e/x-expand to the original token in the token list or to the cluster of tokens corresponding to one Unicode character in the given encoding (for l3str-convert). The ⟨*catcode*⟩ is given as a single hexadecimal digit, 0 for control sequences. The ⟨*char code*⟩ is given as a decimal number, −1 for control sequences.

Using delimited arguments lets us build the ⟨*tokens*⟩ progressively when doing an encoding conversion in l3str-convert. On the other hand, the delimiter `\s__tl` may not appear unbraced in ⟨*tokens*⟩. This is not a problem because we are careful to wrap control sequences in braces (as an argument to `\exp_not:n`) when converting from a general token list to the internal format.

The current rule for converting a ⟨*token*⟩ to a balanced set of ⟨*tokens*⟩ which both o-expands and e/x-expands to it is the following.

- A control sequence `\cs` becomes `\exp_not:n { \cs } \s__tl` 0 -1 `\s__tl`.
- A begin-group character `{` becomes `\exp_after:wN { \if_false: } \fi: \s__tl` 1 ⟨*char code*⟩ `\s__tl`.
- An end-group character `}` becomes `\if_false: { \fi: } \s__tl` 2 ⟨*char code*⟩ `\s__tl`.
- A character with any other category code becomes `\exp_not:n` {⟨*character*⟩} `\s__tl` ⟨*hex catcode*⟩ ⟨*char code*⟩ `\s__tl`.

In contrast, for `\peek_analysis_map_inline:n` we must allow for an input stream containing `\outer` macros, so that wrapping all control sequences in `\exp_not:n` is unsafe. Instead, we write the more elaborate `\__kernel_exp_not:w \exp_after:wN { \exp_not:N \cs }`. (On the other hand we make a better effort by avoiding `\exp_not:n` for characters other than active and macro parameters.)

```
⟨*package⟩
```

45.3 Variables and helper functions

`\s__tl`

The scan mark `\s__tl` is used as a delimiter in the internal format. This is more practical than using a quark, because we would then need to control expansion much more carefully: compare `\int_value:w '#1 \s__tl` with `\int_value:w '#1 \exp_stop_f: \exp_not:N \q_mark` to extract a character code followed by the delimiter in an e-expansion.

```
\scan_new:N \s__tl
```

(*End of definition for* `\s__tl`.)

`\l__tl_analysis_token`
`\l__tl_analysis_char_token`

The tokens in the token list are probed with the TeX primitive `\futurelet`. We use `\l__tl_analysis_token` in that construction. In some cases, we convert the following token to a string before probing it: then the token variable used is `\l__tl_analysis_char_token`.

```
\cs_new_eq:NN \l__tl_analysis_token ?
\cs_new_eq:NN \l__tl_analysis_char_token ?
```

(*End of definition for* `\l__tl_analysis_token` *and* `\l__tl_analysis_char_token`.)

`\l__tl_peek_code_tl`

Holds some code to be run once the next token has been fully analysed in `\peek_analysis_map_inline:n`.

```
\tl_new:N \l__tl_peek_code_tl
```

(*End of definition for* `\l__tl_peek_code_tl`.)

`\c__tl_peek_catcodes_tl`

A token list containing the character number 32 (space) with all possible category codes except 1 and 2 (begin-group and end-group). Why 32? Because some LuaTeX versions only allow creation of catcode 10 (space) tokens with this character code, so that we decided to make `\char_generate:nn` refuse to create such weird spaces as well. We do not include the macro parameter case (catcode 6) because it cannot be used as a macro delimiter.

```
\group_begin:
\char_set_active_eq:NN \  \scan_stop:
\tl_const:Ne \c__tl_peek_catcodes_tl
```

```
  {
    \char_generate:nn { 32 } { 3 }   3
    \char_generate:nn { 32 } { 4 }   4
    \char_generate:nn { 32 } { 7 }   7
    \char_generate:nn { 32 } { 8 }   8
    \c_space_tl                      \token_to_str:N A
    \char_generate:nn { 32 } { 11 } \token_to_str:N B
    \char_generate:nn { 32 } { 12 } \token_to_str:N C
    \char_generate:nn { 32 } { 13 } \token_to_str:N D
  }
\group_end:
```

(*End of definition for* `\c__tl_peek_catcodes_tl`*.*)

`\l__tl_analysis_normal_int` The number of normal (`N`-type argument) tokens since the last special token.

```
\int_new:N \l__tl_analysis_normal_int
```

(*End of definition for* `\l__tl_analysis_normal_int`*.*)

`\l__tl_analysis_index_int` During the first pass, this is the index in the array being built. During the second pass, it is equal to the maximum index in the array from the first pass.

```
\int_new:N \l__tl_analysis_index_int
```

(*End of definition for* `\l__tl_analysis_index_int`*.*)

`\l__tl_analysis_nesting_int` Nesting depth of explicit begin-group and end-group characters during the first pass. This lets us detect the end of the token list without a reserved end-marker.

```
\int_new:N \l__tl_analysis_nesting_int
```

(*End of definition for* `\l__tl_analysis_nesting_int`*.*)

`\l__tl_analysis_type_int` When encountering special characters, we record their "type" in this integer.

```
\int_new:N \l__tl_analysis_type_int
```

(*End of definition for* `\l__tl_analysis_type_int`*.*)

`\g__tl_analysis_result_tl` The result of the conversion is stored in this token list, with a succession of items of the form

⟨*tokens*⟩ `\s__tl` ⟨*catcode*⟩ ⟨*char code*⟩ `\s__tl`

```
\tl_new:N \g__tl_analysis_result_tl
```

(*End of definition for* `\g__tl_analysis_result_tl`*.*)

`\__tl_analysis_extract_charcode:` `\__tl_analysis_extract_charcode_aux:w` Extracting the character code from the meaning of `\l__tl_analysis_token`. This has no error checking, and should only be assumed to work for begin-group and end-group character tokens. It produces a number in the form `‘`⟨*char*⟩.

```
\cs_new:Npn \__tl_analysis_extract_charcode:
  {
    \exp_after:wN \__tl_analysis_extract_charcode_aux:w
      \token_to_meaning:N \l__tl_analysis_token
  }
\cs_new:Npn \__tl_analysis_extract_charcode_aux:w #1 ~ #2 ~ { ‘ }
```

(*End of definition for* `\__tl_analysis_extract_charcode:` *and* `\__tl_analysis_extract_charcode_aux:w`.)

__tl_analysis_cs_space_count:NN
__tl_analysis_cs_space_count:w
__tl_analysis_cs_space_count_end:w

Counts the number of spaces in the string representation of its second argument, as well as the number of characters following the last space in that representation, and feeds the two numbers as semicolon-delimited arguments to the first argument. When this function is used, the escape character is printable and non-space.

```
\cs_new:Npn \__tl_analysis_cs_space_count:NN #1 #2
  {
    \exp_after:wN #1
    \int_value:w \int_eval:w 0
      \exp_after:wN \__tl_analysis_cs_space_count:w
        \token_to_str:N #2
        \fi: \__tl_analysis_cs_space_count_end:w ; ~ !
  }
\cs_new:Npn \__tl_analysis_cs_space_count:w #1 ~
  {
    \if_false: #1 #1 \fi:
    + 1
    \__tl_analysis_cs_space_count:w
  }
\cs_new:Npn \__tl_analysis_cs_space_count_end:w ; #1 \fi: #2 !
  { \exp_after:wN ; \int_value:w \str_count_ignore_spaces:n {#1} ; }
```

(*End of definition for* `\__tl_analysis_cs_space_count:NN`*,* `\__tl_analysis_cs_space_count:w`*, and* `\__tl_analysis_cs_space_count_end:w`.)

45.4 Plan of attack

Our goal is to produce a token list of the form roughly

⟨*token 1*⟩ `\s__tl` ⟨*catcode 1*⟩ ⟨*char code 1*⟩ `\s__tl`
⟨*token 2*⟩ `\s__tl` ⟨*catcode 2*⟩ ⟨*char code 2*⟩ `\s__tl`
... ⟨*token N*⟩ `\s__tl` ⟨*catcode N*⟩ ⟨*char code N*⟩ `\s__tl`

Most but not all tokens can be grabbed as an undelimited (`N`-type) argument by TeX. The plan is to have a two pass system. In the first pass, locate special tokens, and store them in various `\toks` registers. In the second pass, which is done within an `e`-expanding assignment, normal tokens are taken in as `N`-type arguments, and special tokens are retrieved from the `\toks` registers, and removed from the input stream by some means. The whole process takes linear time, because we avoid building the result one item at a time.

We make the escape character printable (backslash, but this later oscillates between slash and backslash): this allows us to distinguish characters from control sequences.

A token has two characteristics: its `\meaning`, and what it looks like for TeX when it is in scanning mode (*e.g.*, when capturing parameters for a macro). For our purposes, we distinguish the following meanings:

- begin-group token (category code 1), either space (character code 32), or non-space;
- end-group token (category code 2), either space (character code 32), or non-space;
- space token (category code 10, character code 32);

- anything else (then the token is always an N-type argument).

The token itself can "look like" one of the following

- a non-active character, in which case its meaning is automatically that associated to its character code and category code, we call it "true" character;
- an active character;
- a control sequence.

The only tokens which are not valid N-type arguments are true begin-group characters, true end-group characters, and true spaces. We detect those characters by scanning ahead with `\futurelet`, then distinguishing true characters from control sequences set equal to them using the `\string` representation.

The second pass is a simple exercise in expandable loops.

`\__tl_analysis:n` Everything is done within a group, and all definitions are local. We use `\group_align_safe_begin/end:` to avoid problems in case `\__tl_analysis:n` is used within an alignment and its argument contains alignment tab tokens.

```
\cs_new_protected:Npn \__tl_analysis:n #1
  {
    \group_begin:
      \group_align_safe_begin:
        \__tl_analysis_a:n {#1}
        \__tl_analysis_b:n {#1}
      \group_align_safe_end:
    \group_end:
  }
```

(*End of definition for* `\__tl_analysis:n`.)

45.5 Disabling active characters

`\__tl_analysis_disable:n` Active characters can cause problems later on in the processing, so we provide a way to disable them, by setting them to `undefined`. Since Unicode contains too many characters to loop over all of them, we instead do this whenever we encounter a character. For pTeX and upTeX we skip characters beyond $[0, 255]$ because `\lccode` only allows those values.

```
\group_begin:
  \char_set_catcode_active:N \^^@
  \cs_new_protected:Npn \__tl_analysis_disable:n #1
    {
      \tex_lccode:D 0 = #1 \exp_stop_f:
      \tex_lowercase:D { \tex_let:D ^^@ } \tex_undefined:D
    }
  \bool_lazy_or:nnT
    { \sys_if_engine_ptex_p: }
    { \sys_if_engine_uptex_p: }
    {
      \cs_gset_protected:Npn \__tl_analysis_disable:n #1
        {
          \if_int_compare:w 256 > #1 \exp_stop_f:
            \tex_lccode:D 0 = #1 \exp_stop_f:
```

```
            \tex_lowercase:D { \tex_let:D ^^@ } \tex_undefined:D
          \fi:
        }
    }
\group_end:
```

(*End of definition for* `\__tl_analysis_disable:n`.)

`\__tl_analysis_disable_char:N` Similar to `\__tl_analysis_disable:n`, but it receives a normal character token, tests if that token is active (by turning it into a space: the active space has been undefined at this point), and if so, disables it. Even if the character is active and set equal to a primitive conditional, nothing blows up. Again, in pTEX and upTEX we skip characters beyond $[0, 255]$, which cannot be active anyways.

```
\group_begin:
  \char_set_catcode_active:N \^^@
  \cs_new_protected:Npn \__tl_analysis_disable_char:N #1
    {
      \tex_lccode:D `#1 = 32 \exp_stop_f:
      \tex_lowercase:D { \if_meaning:w #1 } \tex_undefined:D
        \tex_let:D #1 \tex_undefined:D
      \fi:
    }
  \bool_lazy_or:nnT
    { \sys_if_engine_ptex_p: }
    { \sys_if_engine_uptex_p: }
    {
      \cs_gset_protected:Npn \__tl_analysis_disable_char:N #1
        {
          \if_int_compare:w 256 > `#1 \exp_stop_f:
            \tex_lccode:D `#1 = 32 \exp_stop_f:
            \tex_lowercase:D { \if_meaning:w #1 } \tex_undefined:D
              \tex_let:D #1 \tex_undefined:D
            \fi:
          \fi:
        }
    }
\group_end:
```

(*End of definition for* `\__tl_analysis_disable_char:N`.)

45.6 First pass

The goal of this pass is to detect special (non-`N`-type) tokens, and count how many `N`-type tokens lie between special tokens. Also, we wish to store some representation of each special token in a `\toks` register.

We have 11 types of tokens:

1. a true non-space begin-group character;
2. a true space begin-group character;
3. a true non-space end-group character;
4. a true space end-group character;

5. a true space blank space character;
6. an active character;
7. any other true character;
8. a control sequence equal to a begin-group token (category code 1);
9. a control sequence equal to an end-group token (category code 2);
10. a control sequence equal to a space token (character code 32, category code 10);
11. any other control sequence.

Our first tool is `\futurelet`. This cannot distinguish case 8 from 1 or 2, nor case 9 from 3 or 4, nor case 10 from case 5. Those cases are later distinguished by applying the `\string` primitive to the following token, after possibly changing the escape character to ensure that a control sequence's string representation cannot be mistaken for the true character.

In cases 6, 7, and 11, the following token is a valid `N`-type argument, so we grab it and distinguish the case of a character from a control sequence: in the latter case, `\str_tail:n {`⟨*token*⟩`}` is non-empty, because the escape character is printable.

`\__tl_analysis_a:n` We read tokens one by one using `\futurelet`. While performing the loop, we keep track of the number of true begin-group characters minus the number of true end-group characters in `\l__tl_analysis_nesting_int`. This reaches -1 when we read the closing brace.

```
\cs_new_protected:Npn \__tl_analysis_a:n #1
  {
    \__tl_analysis_disable:n { 32 }
    \int_set:Nn \tex_escapechar:D { 92 }
    \int_zero:N \l__tl_analysis_normal_int
    \int_zero:N \l__tl_analysis_index_int
    \int_zero:N \l__tl_analysis_nesting_int
    \if_false: { \fi: \__tl_analysis_a_loop:w #1 }
    \int_decr:N \l__tl_analysis_index_int
  }
```

(*End of definition for* `\__tl_analysis_a:n`.)

`\__tl_analysis_a_loop:w` Read one character and check its type.

```
\cs_new_protected:Npn \__tl_analysis_a_loop:w
  { \tex_futurelet:D \l__tl_analysis_token \__tl_analysis_a_type:w }
```

(*End of definition for* `\__tl_analysis_a_loop:w`.)

`\__tl_analysis_a_type:w` At this point, `\l__tl_analysis_token` holds the meaning of the following token. We store in `\l__tl_analysis_type_int` information about the meaning of the token ahead:

- 0 space token;
- 1 begin-group token;
- -1 end-group token;
- 2 other.

The values 0, 1, −1 correspond to how much a true such character changes the nesting level (2 is used only here, and is irrelevant later). Then call the auxiliary for each case. Note that nesting conditionals here is safe because we only skip over `\l__tl_analysis_token` if it matches with one of the character tokens (hence is not a primitive conditional).

```
\cs_new_protected:Npn \__tl_analysis_a_type:w
  {
    \l__tl_analysis_type_int =
      \if_meaning:w \l__tl_analysis_token \c_space_token
        0
      \else:
        \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_begin_token
          1
        \else:
          \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_end_token
            - 1
          \else:
            2
          \fi:
        \fi:
      \fi:
      \exp_stop_f:
    \if_case:w \l__tl_analysis_type_int
         \exp_after:wN \__tl_analysis_a_space:w
    \or: \exp_after:wN \__tl_analysis_a_bgroup:w
    \or: \exp_after:wN \__tl_analysis_a_safe:N
    \else: \exp_after:wN \__tl_analysis_a_egroup:w
    \fi:
  }
```

(*End of definition for* `\__tl_analysis_a_type:w`.)

`\__tl_analysis_a_space:w` `\__tl_analysis_a_space_test:w`

In this branch, the following token's meaning is a blank space. Apply `\string` to that token: a true blank space gives a space, a control sequence gives a result starting with the escape character, an active character gives something else than a space since we disabled the space. We grab as `\l__tl_analysis_char_token` the first character of the string representation then test it in `\__tl_analysis_a_space_test:w`. Also, since `\__tl_analysis_a_store:` expects the special token to be stored in the relevant `\toks` register, we do that. The extra `\exp_not:n` is unnecessary of course, but it makes the treatment of all tokens more homogeneous. If we discover that the next token was actually a control sequence or an active character instead of a true space, then we step the counter of normal tokens. We now have in front of us the whole string representation of the control sequence, including potential spaces; those will appear to be true spaces later in this pass. Hence, all other branches of the code in this first pass need to consider the string representation, so that the second pass does not need to test the meaning of tokens, only strings.

```
\cs_new_protected:Npn \__tl_analysis_a_space:w
  {
    \tex_afterassignment:D \__tl_analysis_a_space_test:w
    \exp_after:wN \cs_set_eq:NN
    \exp_after:wN \l__tl_analysis_char_token
    \token_to_str:N
  }
\cs_new_protected:Npn \__tl_analysis_a_space_test:w
```

```
  {
    \if_meaning:w \l__tl_analysis_char_token \c_space_token
      \tex_toks:D \l__tl_analysis_index_int { \exp_not:n { ~ } }
      \__tl_analysis_a_store:
    \else:
      \int_incr:N \l__tl_analysis_normal_int
    \fi:
    \__tl_analysis_a_loop:w
  }
```

(*End of definition for* `\__tl_analysis_a_space:w` *and* `\__tl_analysis_a_space_test:w`.)

`\__tl_analysis_a_bgroup:w`
`\__tl_analysis_a_egroup:w`
`\__tl_analysis_a_group:nw`
`\__tl_analysis_a_group_aux:w`
`\__tl_analysis_a_group_auxii:w`
`\__tl_analysis_a_group_test:w`

The token is most likely a true character token with catcode 1 or 2, but it might be a control sequence, or an active character. Optimizing for the first case, we store in a toks register some code that expands to that token. Since we will turn what follows into a string, we make sure the escape character is different from the current character code (by switching between solidus and backslash). To detect the special case of an active character let to the catcode 1 or 2 character with the same character code, we disable the active character with that character code and re-test: if the following token has become undefined we can in fact safely grab it. We are finally ready to turn what follows to a string and test it. This is one place where we need `\l__tl_analysis_char_token` to be a separate control sequence from `\l__tl_analysis_token`, to compare them.

```
\group_begin:
  \char_set_catcode_group_begin:N \^^@ % {
  \cs_new_protected:Npn \__tl_analysis_a_bgroup:w
    { \__tl_analysis_a_group:nw { \exp_after:wN ^^@ \if_false: } \fi: } }
  \char_set_catcode_group_end:N \^^@
  \cs_new_protected:Npn \__tl_analysis_a_egroup:w
    { \__tl_analysis_a_group:nw { \if_false: { \fi: ^^@ } } % }
\group_end:
\cs_new_protected:Npn \__tl_analysis_a_group:nw #1
  {
    \tex_lccode:D 0 = \__tl_analysis_extract_charcode: \scan_stop:
    \tex_lowercase:D { \tex_toks:D \l__tl_analysis_index_int {#1} }
    \if_int_compare:w \tex_lccode:D 0 = \tex_escapechar:D
      \int_set:Nn \tex_escapechar:D { 139 - \tex_escapechar:D }
    \fi:
    \__tl_analysis_disable:n { \tex_lccode:D 0 }
    \tex_futurelet:D \l__tl_analysis_token \__tl_analysis_a_group_aux:w
  }
\cs_new_protected:Npn \__tl_analysis_a_group_aux:w
  {
    \if_meaning:w \l__tl_analysis_token \tex_undefined:D
      \exp_after:wN \__tl_analysis_a_safe:N
    \else:
      \exp_after:wN \__tl_analysis_a_group_auxii:w
    \fi:
  }
\cs_new_protected:Npn \__tl_analysis_a_group_auxii:w
  {
    \tex_afterassignment:D \__tl_analysis_a_group_test:w
    \exp_after:wN \cs_set_eq:NN
    \exp_after:wN \l__tl_analysis_char_token
    \token_to_str:N
```

```
  }
\cs_new_protected:Npn \__tl_analysis_a_group_test:w
  {
    \if_charcode:w \l__tl_analysis_token \l__tl_analysis_char_token
      \__tl_analysis_a_store:
    \else:
      \int_incr:N \l__tl_analysis_normal_int
    \fi:
    \__tl_analysis_a_loop:w
  }
```

(*End of definition for* `\__tl_analysis_a_bgroup:w` *and others.*)

`\__tl_analysis_a_store:` This function is called each time we meet a special token; at this point, the `\toks` register `\l__tl_analysis_index_int` holds a token list which expands to the given special token. Also, the value of `\l__tl_analysis_type_int` indicates which case we are in:

- -1 end-group character;
- 0 space character;
- 1 begin-group character.

We need to distinguish further the case of a space character (code 32) from other character codes, because those behave differently in the second pass. Namely, after testing the `\lccode` of 0 (which holds the present character code) we change the cases above to

- -2 space end-group character;
- -1 non-space end-group character;
- 0 space blank space character;
- 1 non-space begin-group character;
- 2 space begin-group character.

This has the property that non-space characters correspond to odd values of `\l__tl_analysis_type_int`. The number of normal tokens until here and the type of special token are packed into a `\skip` register. Finally, we check whether we reached the last closing brace, in which case we stop by disabling the looping function (locally).

```
\cs_new_protected:Npn \__tl_analysis_a_store:
  {
    \tex_advance:D \l__tl_analysis_nesting_int \l__tl_analysis_type_int
    \if_int_compare:w \tex_lccode:D 0 = `\ \exp_stop_f:
      \tex_advance:D \l__tl_analysis_type_int \l__tl_analysis_type_int
    \fi:
    \tex_skip:D \l__tl_analysis_index_int
      = \l__tl_analysis_normal_int sp
         plus \l__tl_analysis_type_int sp \scan_stop:
    \int_incr:N \l__tl_analysis_index_int
    \int_zero:N \l__tl_analysis_normal_int
    \if_int_compare:w \l__tl_analysis_nesting_int = - \c_one_int
      \cs_set_eq:NN \__tl_analysis_a_loop:w \scan_stop:
    \fi:
  }
```

(*End of definition for* `\__tl_analysis_a_store:`.)

`\__tl_analysis_a_safe:N`
`\__tl_analysis_a_cs:ww`

This should be the simplest case: since the upcoming token is safe, we can simply grab it in a second pass. If the token is a single character (including space), the `\if_charcode:w` test yields true; we disable a potentially active character (that could otherwise masquerade as the true character in the next pass) and we count one "normal" token. On the other hand, if the token is a control sequence, we should replace it by its string representation for compatibility with other code branches. Instead of slowly looping through the characters with the main code, we use the knowledge of how the second pass works: if the control sequence name contains no space, count that token as a number of normal tokens equal to its string length. If the control sequence contains spaces, they should be registered as special characters by increasing `\l__tl_analysis_index_int` (no need to carefully count character between each space), and all characters after the last space should be counted in the following sequence of "normal" tokens.

```
\cs_new_protected:Npn \__tl_analysis_a_safe:N #1
  {
    \if_charcode:w
        \scan_stop:
        \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
        \scan_stop:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
      {
        \__tl_analysis_disable_char:N #1
        \int_incr:N \l__tl_analysis_normal_int
      }
      { \__tl_analysis_cs_space_count:NN \__tl_analysis_a_cs:ww #1 }
    \__tl_analysis_a_loop:w
  }
\cs_new_protected:Npn \__tl_analysis_a_cs:ww #1; #2;
  {
    \if_int_compare:w #1 > \c_zero_int
      \tex_skip:D \l__tl_analysis_index_int
        = \int_eval:n { \l__tl_analysis_normal_int + 1 } sp \exp_stop_f:
      \tex_advance:D \l__tl_analysis_index_int #1 \exp_stop_f:
    \else:
      \tex_advance:D
    \fi:
    \l__tl_analysis_normal_int #2 \exp_stop_f:
  }
```

(*End of definition for* `\__tl_analysis_a_safe:N` *and* `\__tl_analysis_a_cs:ww`.)

45.7 Second pass

The second pass is an exercise in expandable loops. All the necessary information is stored in `\skip` and `\toks` registers.

`\__tl_analysis_b:n`
`\__tl_analysis_b_loop:w`

Start the loop with the index 0. No need for an end-marker: the loop stops by itself when the last index is read. We repeatedly oscillate between reading long stretches of normal tokens, and reading special tokens.

```
\cs_new_protected:Npn \__tl_analysis_b:n #1
  {
    \__kernel_tl_gset:Nx \g__tl_analysis_result_tl
      {
        \__tl_analysis_b_loop:w 0; #1
        \prg_break_point:
      }
  }
\cs_new:Npn \__tl_analysis_b_loop:w #1;
  {
    \exp_after:wN \__tl_analysis_b_normals:ww
      \int_value:w \tex_skip:D #1 ; #1 ;
  }
```

(*End of definition for* `\__tl_analysis_b:n` *and* `\__tl_analysis_b_loop:w`.)

`\__tl_analysis_b_normals:ww`
`\__tl_analysis_b_normal:wwN`

The first argument is the number of normal tokens which remain to be read, and the second argument is the index in the array produced in the first step. A character's string representation is always one character long, while a control sequence is always longer (we have set the escape character to a printable value). In both cases, we leave `\exp_not:n {`⟨*token*⟩`} \s__tl` in the input stream (after `e`-expansion). Here, `\exp_not:n` is used rather than `\exp_not:N` because `#3` could be a macro parameter character or could be `\s__tl` (which must be hidden behind braces in the result).

```
\cs_new:Npn \__tl_analysis_b_normals:ww #1;
  {
    \if_int_compare:w #1 = \c_zero_int
      \__tl_analysis_b_special:w
    \fi:
    \__tl_analysis_b_normal:wwN #1;
  }
\cs_new:Npn \__tl_analysis_b_normal:wwN #1; #2; #3
  {
    \exp_not:n { \exp_not:n { #3 } } \s__tl
    \if_charcode:w
        \scan_stop:
        \exp_after:wN \use_none:n \token_to_str:N #3 \prg_do_nothing:
        \scan_stop:
      \exp_after:wN \__tl_analysis_b_char:Nn
      \exp_after:wN \__tl_analysis_b_char_aux:nww
    \else:
      \exp_after:wN \__tl_analysis_b_cs:Nww
    \fi:
    #3 #1; #2;
  }
```

(*End of definition for* `\__tl_analysis_b_normals:ww` *and* `\__tl_analysis_b_normal:wwN`.)

`\__tl_analysis_b_char:Nn`
`\__tl_analysis_b_char_aux:nww`

This function is called here with arguments `\__tl_analysis_b_char_aux:nww` and a normal character, while in the peek analysis code it is called with `\use_none:n` and possibly a space character, which is why the function has signature `Nn`. If the normal token we grab is a character, leave ⟨*catcode*⟩ ⟨*charcode*⟩ followed by `\s__tl` in the input stream, and call `\__tl_analysis_b_normals:ww` with its first argument decremented.

```
\cs_new:Npe \__tl_analysis_b_char:Nn #1#2
  {
```

```
    \exp_not:N \if_meaning:w #2 \exp_not:N \tex_undefined:D
      \token_to_str:N D \exp_not:N \else:
    \exp_not:N \if_catcode:w #2 \c_catcode_other_token
      \token_to_str:N C \exp_not:N \else:
    \exp_not:N \if_catcode:w #2 \c_catcode_letter_token
      \token_to_str:N B \exp_not:N \else:
    \exp_not:N \if_catcode:w #2 \c_math_toggle_token      3
      \exp_not:N \else:
    \exp_not:N \if_catcode:w #2 \c_alignment_token        4
      \exp_not:N \else:
    \exp_not:N \if_catcode:w #2 \c_math_superscript_token 7
      \exp_not:N \else:
    \exp_not:N \if_catcode:w #2 \c_math_subscript_token   8
      \exp_not:N \else:
    \exp_not:N \if_catcode:w #2 \c_space_token
      \token_to_str:N A \exp_not:N \else:
      6
    \exp_not:n { \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi: }
    #1 {#2}
  }
\cs_new:Npn \__tl_analysis_b_char_aux:nww #1
  {
    \int_value:w `#1 \s__tl
    \exp_after:wN \__tl_analysis_b_normals:ww
      \int_value:w \int_eval:w - 1 +
  }
```

(*End of definition for* `\__tl_analysis_b_char:Nn` *and* `\__tl_analysis_b_char_aux:nww`.)

`\__tl_analysis_b_cs:Nww`
`\__tl_analysis_b_cs_test:ww`

If the token we grab is a control sequence, leave `0 -1` (as category code and character code) in the input stream, followed by `\s__tl`, and call `\__tl_analysis_b_normals:ww` with updated arguments.

```
\cs_new:Npn \__tl_analysis_b_cs:Nww #1
  {
    0 -1 \s__tl
    \__tl_analysis_cs_space_count:NN \__tl_analysis_b_cs_test:ww #1
  }
\cs_new:Npn \__tl_analysis_b_cs_test:ww #1 ; #2 ; #3 ; #4 ;
  {
    \exp_after:wN \__tl_analysis_b_normals:ww
    \int_value:w \int_eval:w
    \if_int_compare:w #1 = \c_zero_int
      #3
    \else:
      \tex_skip:D \int_eval:n { #4 + #1 } \exp_stop_f:
    \fi:
    - #2
    \exp_after:wN ;
    \int_value:w \int_eval:n { #4 + #1 } ;
  }
```

(*End of definition for* `\__tl_analysis_b_cs:Nww` *and* `\__tl_analysis_b_cs_test:ww`.)

`\__tl_analysis_b_special:w`
`\__tl_analysis_b_special_char:wN`
`\__tl_analysis_b_special_space:w`

Here, `#1` is the current index in the array built in the first pass. Check now whether we reached the end (we shouldn't keep the trailing end-group character that marked the end of the token list in the first pass). Unpack the `\toks` register: when e/x-expanding again, we will get the special token. Then leave the category code in the input stream, followed by the character code, and call `\__tl_analysis_b_loop:w` with the next index.

```
\group_begin:
  \char_set_catcode_other:N A
  \cs_new:Npn \__tl_analysis_b_special:w
      \fi: \__tl_analysis_b_normal:wwN 0 ; #1 ;
    {
      \fi:
      \if_int_compare:w #1 = \l__tl_analysis_index_int
        \exp_after:wN \prg_break:
      \fi:
      \tex_the:D \tex_toks:D #1 \s__tl
      \if_case:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
             \token_to_str:N A
      \or:   1
      \or:   1
      \else: 2
      \fi:
      \if_int_odd:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
        \exp_after:wN \__tl_analysis_b_special_char:wN \int_value:w
      \else:
        \exp_after:wN \__tl_analysis_b_special_space:w \int_value:w
      \fi:
      \int_eval:n { 1 + #1 } \exp_after:wN ;
      \token_to_str:N
    }
\group_end:
\cs_new:Npn \__tl_analysis_b_special_char:wN #1 ; #2
  {
    \int_value:w `#2 \s__tl
    \__tl_analysis_b_loop:w #1 ;
  }
\cs_new:Npn \__tl_analysis_b_special_space:w #1 ; ~
  {
    32 \s__tl
    \__tl_analysis_b_loop:w #1 ;
  }
```

(*End of definition for* `\__tl_analysis_b_special:w`*,* `\__tl_analysis_b_special_char:wN`*, and* `\__tl_analysis_b_special_space:w`*.*)

45.8 Mapping through the analysis

`\tl_analysis_map_inline:Nn`
`\tl_analysis_map_inline:nn`
`\__tl_analysis_map:Nn`
`\__tl_analysis_map:NwNw`

First obtain the analysis of the token list into `\g__tl_analysis_result_tl`. To allow nested mappings, increase the nesting depth `\g__kernel_prg_map_int` (shared between all modules), then define the payload macro, which runs the user code and has a name specific to that nesting depth. The looping macro grabs the ⟨*tokens*⟩, ⟨*catcode*⟩ and ⟨*char code*⟩; it checks for the end of the loop with `\use_none:n ##2`, normally empty, but which becomes `\tl_map_break:` at the end; it then calls the payload macro with the arguments in the correct order (this is the reason why we cannot directly use the same macro for looping and payload), and loops by calling itself. When the loop ends, remember to decrease the nesting depth.

```
\cs_new_protected:Npn \tl_analysis_map_inline:Nn #1
  { \exp_args:No \tl_analysis_map_inline:nn #1 }
\cs_new_protected:Npn \tl_analysis_map_inline:nn #1
  {
    \__tl_analysis:n {#1}
    \int_gincr:N \g__kernel_prg_map_int
    \exp_args:Nc \__tl_analysis_map:Nn
      { __tl_analysis_map_inline_ \int_use:N \g__kernel_prg_map_int :wNw }
  }
\cs_new_protected:Npn \__tl_analysis_map:Nn #1#2
  {
    \cs_gset_protected:Npn #1 ##1##2##3 {#2}
    \exp_after:wN \__tl_analysis_map:NwNw \exp_after:wN #1
      \g__tl_analysis_result_tl
      \s__tl { ? \tl_map_break: } \s__tl
    \prg_break_point:Nn \tl_map_break:
      { \int_gdecr:N \g__kernel_prg_map_int }
  }
\cs_new_protected:Npn \__tl_analysis_map:NwNw #1 #2 \s__tl #3 #4 \s__tl
  {
    \use_none:n #3
    #1 {#2} {#4} {#3}
    \__tl_analysis_map:NwNw #1
  }
```

(*End of definition for `\tl_analysis_map_inline:Nn` and others. These functions are documented on page 47.*)

45.9 Showing the results

`\tl_analysis_show:N`
`\tl_analysis_log:N`
`\__tl_analysis_show:NNN`

Add to `\__tl_analysis:n` a third pass to display tokens to the terminal. If the token list variable is not defined, throw the same error as `\tl_show:N` by simply calling that function.

```
\cs_new_protected:Npn \tl_analysis_show:N
  { \__tl_analysis_show:NNN \msg_show:nneeee \tl_show:N }
\cs_new_protected:Npn \tl_analysis_log:N
  { \__tl_analysis_show:NNN \msg_log:nneeee \tl_log:N }
\cs_new_protected:Npn \__tl_analysis_show:NNN #1#2#3
  {
    \tl_if_exist:NTF #3
      {
        \exp_args:No \__tl_analysis:n {#3}
        #1 { tl } { show-analysis }
          { \token_to_str:N #3 } { \__tl_analysis_show: } { } { }
      }
      { #2 #3 }
  }
```

(*End of definition for `\tl_analysis_show:N`, `\tl_analysis_log:N`, and `\__tl_analysis_show:NNN`. These functions are documented on page 47.*)

\tl_analysis_show:n \tl_analysis_log:n __tl_analysis_show:Nn

No existence test needed here.

```
\cs_new_protected:Npn \tl_analysis_show:n
  { \__tl_analysis_show:Nn \msg_show:nneeee }
\cs_new_protected:Npn \tl_analysis_log:n
  { \__tl_analysis_show:Nn \msg_log:nneeee }
\cs_new_protected:Npn \__tl_analysis_show:Nn #1#2
  {
    \__tl_analysis:n {#2}
    #1 { tl } { show-analysis } { } { \__tl_analysis_show: } { } { }
  }
```

(*End of definition for* \tl_analysis_show:n *,* \tl_analysis_log:n *, and* __tl_analysis_show:Nn*. These functions are documented on page* 47*.*)

__tl_analysis_show: __tl_analysis_show_loop:wNw

Here, #1 o- and e/x-expands to the token; #2 is the category code (one uppercase hexadecimal digit), 0 for control sequences; #3 is the character code, which we ignore. In the cases of control sequences and active characters, the meaning may overflow one line, and we want to truncate it. Those cases are thus separated out.

```
\cs_new:Npn \__tl_analysis_show:
  {
    \exp_after:wN \__tl_analysis_show_loop:wNw \g__tl_analysis_result_tl
    \s__tl { ? \prg_break: } \s__tl
    \prg_break_point:
  }
\cs_new:Npn \__tl_analysis_show_loop:wNw #1 \s__tl #2 #3 \s__tl
  {
    \use_none:n #2
    \iow_newline: > \use:nn { ~ } { ~ }
    \if_int_compare:w "#2 = \c_zero_int
      \exp_after:wN \__tl_analysis_show_cs:n
    \else:
      \if_int_compare:w "#2 = 13 \exp_stop_f:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__tl_analysis_show_active:n
      \else:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__tl_analysis_show_normal:n
      \fi:
    \fi:
    {#1}
    \__tl_analysis_show_loop:wNw
  }
```

(*End of definition for* __tl_analysis_show: *and* __tl_analysis_show_loop:wNw*.*)

__tl_analysis_show_normal:n

Non-active characters are a simple matter of printing the character, and its meaning. Our test suite checks that begin-group and end-group characters do not mess up TEX's alignment status.

```
\cs_new:Npn \__tl_analysis_show_normal:n #1
  {
    \exp_after:wN \token_to_str:N #1 ~
    ( \exp_after:wN \token_to_meaning:N #1 )
  }
```

(*End of definition for* `\__tl_analysis_show_normal:n`.)

`\__tl_analysis_show_value:N`

This expands to the value of `#1` if it has any.

```
\cs_new:Npn \__tl_analysis_show_value:N #1
  {
    \token_if_expandable:NF #1
      {
        \token_if_chardef:NTF       #1 \prg_break: { }
        \token_if_mathchardef:NTF   #1 \prg_break: { }
        \token_if_dim_register:NTF  #1 \prg_break: { }
        \token_if_int_register:NTF  #1 \prg_break: { }
        \token_if_skip_register:NTF #1 \prg_break: { }
        \token_if_toks_register:NTF #1 \prg_break: { }
        \use_none:nnn
        \prg_break_point:
        \use:n { \exp_after:wN = \tex_the:D #1 }
      }
  }
```

(*End of definition for* `\__tl_analysis_show_value:N`.)

`\__tl_analysis_show_cs:n`
`\__tl_analysis_show_active:n`
`\__tl_analysis_show_long:nn`
`\__tl_analysis_show_long_aux:nnnn`

Control sequences and active characters are printed in the same way, making sure not to go beyond the `\l_iow_line_count_int`. In case of an overflow, we replace the last characters by `\c__tl_analysis_show_etc_str`.

```
\cs_new:Npn \__tl_analysis_show_cs:n #1
  { \exp_args:No \__tl_analysis_show_long:nn {#1} { control~sequence= } }
\cs_new:Npn \__tl_analysis_show_active:n #1
  { \exp_args:No \__tl_analysis_show_long:nn {#1} { active~character= } }
\cs_new:Npn \__tl_analysis_show_long:nn #1
  {
    \__tl_analysis_show_long_aux:oofn
      { \token_to_str:N #1 }
      { \token_to_meaning:N #1 }
      { \__tl_analysis_show_value:N #1 }
  }
\cs_new:Npn \__tl_analysis_show_long_aux:nnnn #1#2#3#4
  {
    \int_compare:nNnTF
      { \str_count:n { #1 ~ ( #4 #2 #3 ) } }
      > { \l_iow_line_count_int - 3 }
      {
        \str_range:nnn { #1 ~ ( #4 #2 #3 ) } { 1 }
          {
            \l_iow_line_count_int - 3
            - \str_count:N \c__tl_analysis_show_etc_str
          }
        \c__tl_analysis_show_etc_str
      }
      { #1 ~ ( #4 #2 #3 ) }
  }
\cs_generate_variant:Nn \__tl_analysis_show_long_aux:nnnn { oof }
```

(*End of definition for* `\__tl_analysis_show_cs:n` *and others.*)

45.10 Peeking ahead

\peek_analysis_map_break:
\peek_analysis_map_break:n

The break statements use the general `\prg_map_break:Nn`.

```
\cs_new:Npn \peek_analysis_map_break:
  { \prg_map_break:Nn \peek_analysis_map_break: { } }
\cs_new:Npn \peek_analysis_map_break:n
  { \prg_map_break:Nn \peek_analysis_map_break: }
```

(*End of definition for* `\peek_analysis_map_break:` *and* `\peek_analysis_map_break:n`. *These functions are documented on page 207.*)

\l__tl_peek_charcode_int

```
\int_new:N \l__tl_peek_charcode_int
```

(*End of definition for* `\l__tl_peek_charcode_int`.)

__tl_analysis_char_arg:Nw
__tl_analysis_char_arg_aux:Nw

After a call to `\futurelet \l__tl_analysis_token` followed by a stringified character token (either explicit space or catcode other character), grab the argument and pass it to `#1`. We only need to do anything in the case of a space.

```
\cs_new:Npn \__tl_analysis_char_arg:Nw
  {
    \if_meaning:w \l__tl_analysis_token \c_space_token
      \exp_after:wN \__tl_analysis_char_arg_aux:Nw
    \fi:
  }
\cs_new:Npn \__tl_analysis_char_arg_aux:Nw #1 ~ { #1 { ~ } }
```

(*End of definition for* `\__tl_analysis_char_arg:Nw` *and* `\__tl_analysis_char_arg_aux:Nw`.)

\peek_analysis_map_inline:n
__tl_peek_analysis_loop:NNn
__tl_peek_analysis_test:
__tl_peek_analysis_exp:N
__tl_peek_analysis_exp_aux:N
__tl_peek_analysis_nonexp:N
__tl_peek_analysis_cs:N
__tl_peek_analysis_char:N
__tl_peek_analysis_char:w
__tl_peek_analysis_special:
__tl_peek_analysis_retest:
__tl_peek_analysis_str:
__tl_peek_analysis_str:w
__tl_peek_analysis_str:n
__tl_peek_analysis_active_str:n
__tl_peek_analysis_explicit:n
__tl_peek_analysis_escape:
__tl_peek_analysis_collect:w
__tl_peek_analysis_collect:n
__tl_peek_analysis_collect_loop:
__tl_peek_analysis_collect_test:
__tl_peek_analysis_collect_end:NNNN

Save the user's code in a control sequence that is suitable for nested maps. We may wish to pass to this function an `\outer` control sequence or active character; for this we will undefine any expandable token (testing if it is `\outer` is much slower) within a group, closed immediately after the function reads its arguments to avoid affecting the user's code or even our peek code (there is no risk of undefining `\group_end:` itself since that is not expandable). This user's code function also calls the loop auxiliary, and includes the trailing `\prg_break_point:Nn` for when the user wants to stop the loop. The loop auxiliary must remove that break point because it must look at the input stream.

```
\cs_new_protected:Npn \peek_analysis_map_inline:n #1
  {
    \group_align_safe_begin:
    \int_gincr:N \g__kernel_prg_map_int
    \cs_set_protected:cpn
      { __tl_analysis_map_ \int_use:N \g__kernel_prg_map_int :nnN }
      ##1##2##3
      {
        \group_end:
        #1
        \__tl_peek_analysis_loop:NNn
          \prg_break_point:Nn \peek_analysis_map_break:
            {
              \int_gdecr:N \g__kernel_prg_map_int
              \group_align_safe_end:
            }
      }
```

```
    \__tl_peek_analysis_loop:NNn ? ? ?
  }
```

The loop starts a group (closed by the user-code function defined above) with a normalized escape character, and checks if the next token is special or N-type (distinguishing expandable from non-expandable tokens). The test for nonexpandable tokens in `\__tl_peek_analysis_test:` must be done after the tests for begin-group, end-group, and space tokens, in case `\l_peek_token` is either `\outer` or is a primitive TeX conditional, as such tokens cannot be skipped over correctly by conditional code.

```
\cs_new_protected:Npn \__tl_peek_analysis_loop:NNn #1#2#3
  {
    \group_begin:
    \tl_set:Ne \l__tl_peek_code_tl
      {
        \exp_not:c
          { __tl_analysis_map_ \int_use:N \g__kernel_prg_map_int :nnN }
      }
    \int_set:Nn \tex_escapechar:D { `\\ }
    \peek_after:Nw \__tl_peek_analysis_test:
  }
\cs_new_protected:Npn \__tl_peek_analysis_test:
  {
    \if_case:w
      \if_catcode:w \exp_not:N \l_peek_token {   \c_max_int \fi:
      \if_catcode:w \exp_not:N \l_peek_token }   \c_max_int \fi:
      \if_meaning:w \l_peek_token \c_space_token \c_max_int \fi:
      \exp_after:wN \if_meaning:w \exp_not:N \l_peek_token \l_peek_token
        \c_one_int
      \fi:
      \c_zero_int
      \exp_after:wN \exp_after:wN
      \exp_after:wN \__tl_peek_analysis_exp:N
      \exp_after:wN \exp_not:N
    \or:
      \exp_after:wN \__tl_peek_analysis_nonexp:N
    \else:
      \exp_after:wN \__tl_peek_analysis_special:
    \fi:
  }
```

Expandable tokens (which are automatically N-type) can be `\outer` macros, hence the need for `\exp_after:wN` and `\exp_not:N` in the code above, which allows the next function to safely grab the token as an argument. To allow the possibly-`\outer` token `#1` as an argument of the ⟨*user's function*⟩ (which is protected and stored in `\l__tl_peek_code_tl`), we set it equal to a harmless macro. This must be done at the very last minute because `#1` may be some pretty important function such as `\exp_after:wN`. Using a primitive `\cs_set_nopar:Npe` expansion (to avoid `\outer` problems) we set up to run the code `\let #1` ⟨*user's function*⟩ ⟨*user's function*⟩ followed by arguments involving `#1`. Regardless of `#1` (including the user's function itself), the user's function is run. It always starts with `\group_end:`, which has not been redefined since `#1` started out as expandable, and which restores the definition of `#1`.

Then we put the elaborate first argument `\__kernel_exp_not:w \exp_after:wN { \exp_not:N #1 }`: indeed we cannot use `\exp_not:n {#1}` as this breaks for an `\outer` macro and we cannot use `\exp_not:N #1`, as o-expanding this yields a "notexpanded" token equal to (a weird) `\relax`, which would have the wrong value for primitive TeX conditionals such as `\if_meaning:w`.

Then we must add `{-1}0` if the token is a control sequence and `{`⟨*charcode*⟩`}D` otherwise. Distinguishing the two cases is easy: since we have made the escape character printable, `\token_to_str:N` gives at least two characters for a control sequence versus a single one for an active character (possibly being a space, in which case the trailing brace group is taken as the first argument of `\__tl_peek_analysis_exp_aux:Nw`). Importantly, `#1` could be an `\outer` token (as it is only set to `\scan_stop:` at the last minute) but once we apply `\token_to_str:N` we no longer need to worry about it.

```
\cs_new_protected:Npn \__tl_peek_analysis_exp:N #1
  {
    \cs_set_nopar:Npe \l__tl_peek_code_tl
      {
        \tex_let:D \exp_not:N #1 \l__tl_peek_code_tl
        \l__tl_peek_code_tl
        {
          \exp_not:n { \__kernel_exp_not:w \exp_after:wN }
            { \exp_not:N \exp_not:N \exp_not:N #1 }
        }
        \exp_after:wN \__tl_peek_analysis_exp_aux:Nw
        \token_to_str:N #1 { } \s__tl
      }
    \l__tl_peek_code_tl
  }
\cs_new:Npe \__tl_peek_analysis_exp_aux:Nw #1#2 \s__tl
  {
    \exp_not:N \if_meaning:w \scan_stop: #2 \scan_stop:
      { \exp_not:N \int_value:w `#1 ~ } \token_to_str:N D
    \exp_not:N \else:
      { -1 } 0
    \exp_not:N \fi:
  }
```

For normal non-expandable tokens we must distinguish characters (including active ones and macro parameter characters) from control sequences (whose string representation is more than one character because we made the escape character printable). For a control sequence call the user code with suitable arguments, wrapping `#1` within `\exp_not:n` just in case it happens to be equal to a macro parameter character. We do not skip `\exp_not:n` when unnecessary, because this auxiliary is also called in `\__tl_peek_analysis_retest:` where we have changed some control sequences or active characters to `\scan_stop:` temporarily.

```
\cs_new_protected:Npn \__tl_peek_analysis_nonexp:N #1
  {
    \if_charcode:w
        \scan_stop:
        \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
        \scan_stop:
      \exp_after:wN \__tl_peek_analysis_char:N
    \else:
      \exp_after:wN \__tl_peek_analysis_cs:N
    \fi:
    #1
```

```
  }
\cs_new_protected:Npn \__tl_peek_analysis_cs:N #1
  { \l__tl_peek_code_tl { \exp_not:n {#1} } { -1 } 0 }
```

For normal characters we must determine their catcode. The main difficulty is that the character may be an active character masquerading as (i.e., set equal to) itself with a different catcode. Two approaches based on `\lowercase` can detect this. One could make an active character with the same catcode as `#1` and change its definition before testing the catcode of `#1`, but in some Unicode engine this fills up the hash table uselessly. Instead, we lowercase `#1` itself, changing its character code to 32, namely space (because LuaTEX cannot turn catcode 10 characters to anything else than character code 32), then we apply `\__tl_analysis_b_char:Nn`, which detects active characters by comparing them to `\tex_undefined:D`, and we must have undefined the active space (locally) for this test to work. To define `\__tl_peek_analysis_char:N` itself we use an `e`-expanding assignment to get the active space in the right place after making it (just for this definition) unexpandable. Finally `\__tl_peek_analysis_char:w` receives the ⟨*charcode*⟩, ⟨*user function*⟩, ⟨*catcode*⟩, and ⟨*token*⟩, and places the arguments in the correct order. It keeps `\exp_not:n` for macro parameter characters and active characters (the latter could be macro parameter characters, and it seems more uniform to always put `\exp_not:n`), and otherwise eliminates it by expanding once with `\exp_args:NNNo`.

```
\group_begin:
\char_set_active_eq:NN \ \scan_stop:
\cs_new_protected:Npe \__tl_peek_analysis_char:N #1
  {
    \cs_set_eq:NN
      \char_generate:nn { 32 } { 13 }
      \exp_not:N \tex_undefined:D
    \tex_lccode:D `#1 = 32 \exp_stop_f:
    \tex_lowercase:D
      {
        \tl_put_right:Ne \exp_not:N \l__tl_peek_code_tl
          { \exp_not:n { \__tl_analysis_b_char:Nn \use_none:n } {#1} }
      }
    \exp_not:n
      {
        \exp_after:wN \__tl_peek_analysis_char:w
        \int_value:w
      }
      `#1
    \exp_not:n { \exp_after:wN \s__tl \l__tl_peek_code_tl }
    #1
  }
\group_end:
\cs_new_protected:Npn \__tl_peek_analysis_char:w #1 \s__tl #2#3#4
  {
    \if_charcode:w 6 #3
    \else:
      \if_charcode:w D #3
      \else:
        \exp_args:NNNo
      \fi:
    \fi:
    #2 { \exp_not:n {#4} } {#1} #3
```

```
  }
```

For special characters the idea is to eventually act with `\token_to_str:N`, then pick up one by one the characters of this string representation until hitting the token that follows. First determine the character code of (the meaning of) the ⟨*token*⟩ (which we know is a special token), make sure the escape character is different from it, normalize the meanings of two active characters and the empty control sequence, and filter out these cases in `\__tl_peek_analysis_retest:`.

```
\cs_new_protected:Npn \__tl_peek_analysis_special:
  {
    \tex_let:D \l__tl_analysis_token = ~ \l_peek_token
    \int_set:Nn \l__tl_peek_charcode_int
      { \__tl_analysis_extract_charcode: }
    \if_int_compare:w \l__tl_peek_charcode_int = \tex_escapechar:D
      \int_set:Nn \tex_escapechar:D { `\/ }
    \fi:
    \char_set_active_eq:nN { \l__tl_peek_charcode_int } \scan_stop:
    \char_set_active_eq:nN { \tex_escapechar:D } \scan_stop:
    \cs_set_eq:cN { } \scan_stop:
    \tex_futurelet:D \l__tl_analysis_token
    \__tl_peek_analysis_retest:
  }
\cs_new_protected:Npn \__tl_peek_analysis_retest:
  {
    \if_meaning:w \l__tl_analysis_token \scan_stop:
      \exp_after:wN \__tl_peek_analysis_nonexp:N
    \else:
      \exp_after:wN \__tl_peek_analysis_str:
    \fi:
  }
```

At this point we know the meaning of the ⟨*token*⟩ in the input stream is `\l_peek_token`, either a space (32, 10) or a begin-group or end-group token (catcode 1 or 2), and we excluded a few cases that would be difficult later (empty control sequence, active character with the same character code as its meaning or as the escape character). The idea is to apply `\token_to_str:N` to the ⟨*token*⟩ then grab characters (of category code 12 except for spaces that have category code 10) to reconstruct it. In earlier versions of the code we would peek at the ⟨*next token*⟩ that lies after ⟨*token*⟩ in the input stream, which would help us be more accurate in reconstructing the ⟨*token*⟩ case in edge cases (mentioned below), but this had the side-effect of tokenizing the input stream (turning characters into tokens) farther ahead than needed.

We hit the ⟨*token*⟩ with `\token_to_str:N` and start grabbing characters. More precisely, by looking at the first character in the string representation of the ⟨*token*⟩ we distinguish three cases: a stringified control sequence starts with the escape character; for an explicit character we find that same character; for an active character we find anything else (we made sure to exclude the case of an active character whose string representation coincides with the other two cases).

```
\cs_new_protected:Npn \__tl_peek_analysis_str:
  {
    \exp_after:wN \tex_futurelet:D
    \exp_after:wN \l__tl_analysis_token
    \exp_after:wN \__tl_peek_analysis_str:w
    \token_to_str:N
```

```
  }
\cs_new_protected:Npn \__tl_peek_analysis_str:w
  { \__tl_analysis_char_arg:Nw \__tl_peek_analysis_str:n }
\cs_new_protected:Npn \__tl_peek_analysis_str:n #1
  {
    \int_case:nnF { `#1 }
      {
        { \l__tl_peek_charcode_int }
          { \__tl_peek_analysis_explicit:n {#1} }
        { \tex_escapechar:D } { \__tl_peek_analysis_escape: }
      }
      { \__tl_peek_analysis_active_str:n {#1} }
  }
```

When #1 is a stringified active character we pass appropriate arguments to the user's code; thankfully \char_generate:nn can make active characters.

```
\cs_new_protected:Npn \__tl_peek_analysis_active_str:n #1
  {
    \tl_put_right:Ne \l__tl_peek_code_tl
      {
        { \char_generate:nn { `#1 } { 13 } }
        { \int_value:w `#1 }
        \token_to_str:N D
      }
    \l__tl_peek_code_tl
  }
```

When #1 matches the character we had extracted from the meaning of \l_peek_token, the token was an explicit character, which can be a standard space, or a begin-group or end-group character with some character code. In the latter two cases we call \char_generate:nn with suitable arguments and put suitable \if_false: \fi: constructions to make the result balanced and such that o-expanding or e/x-expanding gives back a single (unbalanced) begin-group or end-group character.

```
\cs_new_protected:Npn \__tl_peek_analysis_explicit:n #1
  {
    \tl_put_right:Ne \l__tl_peek_code_tl
      {
        \if_meaning:w \l_peek_token \c_space_token
          { ~ } { 32 } \token_to_str:N A
        \else:
          \if_catcode:w \l_peek_token \c_group_begin_token
            {
              \exp_not:N \exp_after:wN
              \char_generate:nn { `#1 } { 1 }
              \exp_not:N \if_false:
              \if_false: { \fi: }
              \exp_not:N \fi:
            }
            { \int_value:w `#1 }
            1
          \else:
            {
              \exp_not:N \if_false:
              { \if_false: } \fi:
```

```
              \exp_not:N \fi:
              \char_generate:nn { `#1 } { 2 }
            }
            { \int_value:w `#1 }
            2
          \fi:
        \fi:
      }
    \l__tl_peek_code_tl
  }
```

Finally there is the case of a special token whose string representation starts with an escape character, namely the token was a control sequence. In that case we could have grabbed the token directly as an N-type argument, but of course we couldn't know that until we had run all the various tests including stringifying the token. We are thus left with the hard work of picking up one by one the characters in the csname (being careful about spaces), until the constructed csname has the expected meaning. This fails if someone defines a token like `\bgroup@my` whose string representation starts the same as another token with the same meaning being an implicit character token of category code 1, 2, or 10.

```
\cs_new_protected:Npn \__tl_peek_analysis_escape:
  {
    \tl_clear:N \l__tl_internal_a_tl
    \tex_futurelet:D \l__tl_analysis_token
      \__tl_peek_analysis_collect:w
  }
\cs_new_protected:Npn \__tl_peek_analysis_collect:w
  { \__tl_analysis_char_arg:Nw \__tl_peek_analysis_collect:n }
\cs_new_protected:Npn \__tl_peek_analysis_collect:n #1
  {
    \tl_put_right:Nn \l__tl_internal_a_tl {#1}
    \__tl_peek_analysis_collect_loop:
  }
\cs_new_protected:Npn \__tl_peek_analysis_collect_loop:
  {
    \exp_after:wN \if_meaning:w
      \cs:w
      \if_cs_exist:w \l__tl_internal_a_tl \cs_end:
        \l__tl_internal_a_tl
      \else:
        c_one % anything short
      \fi:
      \cs_end:
      \l_peek_token
      \__tl_peek_analysis_collect_end:NNNN
    \fi:
    \tex_futurelet:D \l__tl_analysis_token
      \__tl_peek_analysis_collect:w
  }
```

As in all other cases, end by calling the user code with suitable arguments (here `#1` is `\fi:`).

```
\cs_new_protected:Npn \__tl_peek_analysis_collect_end:NNNN #1#2#3#4
  {
```

```
    #1
    \tl_put_right:Ne \l__tl_peek_code_tl
      {
        { \exp_not:N \exp_not:n { \exp_not:c { \l__tl_internal_a_tl } } }
        { -1 }
        0
      }
    \l__tl_peek_code_tl
  }
```

(*End of definition for* `\peek_analysis_map_inline:n` *and others. This function is documented on page* *207*.)

45.11 Messages

`\c__tl_analysis_show_etc_str`

When a control sequence (or active character) and its meaning are too long to fit in one line of the terminal, the end is replaced by this token list.

```
\tl_const:Ne \c__tl_analysis_show_etc_str % (
  { \token_to_str:N \ETC.) }
```

(*End of definition for* `\c__tl_analysis_show_etc_str`*.*)

```
\msg_new:nnn { tl } { show-analysis }
  {
    The~token~list~ \tl_if_empty:nF {#1} { #1 ~ }
    \tl_if_empty:nTF {#2}
      { is~empty }
      { contains~the~tokens: #2 }
  }
⟨/package⟩
```

Chapter 46

l3regex implementation

```
⟨*package⟩
⟨@@=regex⟩
```

46.1 Plan of attack

Most regex engines use backtracking. This allows to provide very powerful features (back-references come to mind first), but it is costly, and raises the problem of catastrophic backtracking. Since TEX is not first and foremost a programming language, complicated code tends to run slowly, and we must use faster, albeit slightly more restrictive, techniques, coming from automata theory.

Given a regular expression of n characters, we do the following:

- (Compiling.) Analyse the regex, finding invalid input, and convert it to an internal representation.
- (Building.) Convert the compiled regex to a non-deterministic finite automaton (NFA) with $O(n)$ states which accepts precisely token lists matching that regex.
- (Matching.) Loop through the query token list one token (one "position") at a time, exploring in parallel every possible path ("active thread") through the NFA, considering active threads in an order determined by the quantifiers' greediness.

We use the following vocabulary in the code comments (and in variable names).

- *Group*: index of the capturing group, -1 for non-capturing groups.
- *Position*: each token in the query is labelled by an integer ⟨*position*⟩, with `min_pos` $-$ 1 $\leq$ ⟨*position*⟩ $\leq$ `max_pos`. The lowest and highest positions `min_pos` $-$ 1 and `max_pos` correspond to imaginary begin and end markers (with non-existent category code and character code). `max_pos` is only set quite late in the processing.
- *Query*: the token list to which we apply the regular expression.
- *State*: each state of the NFA is labelled by an integer ⟨*state*⟩ with `min_state` $\leq$ ⟨*state*⟩ $<$ `max_state`.
- *Active thread*: state of the NFA that is reached when reading the query token list for the matching. Those threads are ordered according to the greediness of quantifiers.

- *Step*: used when matching, starts at 0, incremented every time a character is read, and is not reset when searching for repeated matches. The integer `\l__regex_step_int` is a unique id for all the steps of the matching algorithm.

We use l3intarray to manipulate arrays of integers. We also abuse TeX's `\toks` registers, by accessing them directly by number rather than tying them to control sequence using the `\newtoks` allocation functions. Specifically, these arrays and `\toks` are used as follows. When building, `\toks`⟨*state*⟩ holds the tests and actions to perform in the ⟨*state*⟩ of the NFA. When matching,

- `\g__regex_state_active_intarray` holds the last ⟨*step*⟩ in which each ⟨*state*⟩ was active.
- `\g__regex_thread_info_intarray` consists of blocks for each ⟨*thread*⟩ (with `min_thread` ≤ ⟨*thread*⟩ < `max_thread`). Each block has 1+2`\l__regex_capturing_group_int` entries: the ⟨*state*⟩ in which the ⟨*thread*⟩ currently is, followed by the beginnings of all submatches, and then the ends of all submatches. The ⟨*threads*⟩ are ordered starting from the best to the least preferred.
- `\g__regex_submatch_prev_intarray`, `\g__regex_submatch_begin_intarray` and `\g__regex_submatch_end_intarray` hold, for each submatch (as would be extracted by `\regex_extract_all:nnN`), the place where the submatch started to be looked for and its two end-points. For historical reasons, the minimum index is twice `max_state`, and the used registers go up to `\l__regex_submatch_int`. They are organized in blocks of `\l__regex_capturing_group_int` entries, each block corresponding to one match with all its submatches stored in consecutive entries.

When actually building the result,

- `\toks`⟨*position*⟩ holds ⟨*tokens*⟩ which `o`- and `e`-expand to the ⟨*position*⟩-th token in the query.
- `\g__regex_balance_intarray` holds the balance of begin-group and end-group character tokens which appear before that point in the token list.

The code is structured as follows. Variables are introduced in the relevant section. First we present some generic helper functions. Then comes the code for compiling a regular expression, and for showing the result of the compilation. The building phase converts a compiled regex to NFA states, and the automaton is run by the code in the following section. The only remaining brick is parsing the replacement text and performing the replacement. We are then ready for all the user functions. Finally, messages, and a little bit of tracing code.

46.2 Helpers

`\__regex_int_eval:w`

Access the primitive: performance is key here, so we do not use the slower route *via* `\int_eval:n`.

```
\cs_new_eq:NN \__regex_int_eval:w \tex_numexpr:D
```

(*End of definition for* `\__regex_int_eval:w`.)

`\__regex_standard_escapechar:`

Make the `\escapechar` into the standard backslash.

```
\cs_new_protected:Npn \__regex_standard_escapechar:
  { \int_set:Nn \tex_escapechar:D { `\\ } }
```

(*End of definition for* `\__regex_standard_escapechar:`.)

`\__regex_toks_use:w`

Unpack a `\toks` given its number.

```
\cs_new:Npn \__regex_toks_use:w { \tex_the:D \tex_toks:D }
```

(*End of definition for* `\__regex_toks_use:w`.)

`\__regex_toks_clear:N`
`\__regex_toks_set:Nn`
`\__regex_toks_set:No`

Empty a `\toks` or set it to a value, given its number.

```
\cs_new_protected:Npn \__regex_toks_clear:N #1
  { \tex_toks:D #1 = { } }
\cs_new_eq:NN \__regex_toks_set:Nn \tex_toks:D
\cs_new_protected:Npn \__regex_toks_set:No #1
  { \tex_toks:D #1 = \exp_after:wN }
```

(*End of definition for* `\__regex_toks_clear:N` *and* `\__regex_toks_set:Nn`.)

`\__regex_toks_memcpy:NNn`

Copy `#3` `\toks` registers from `#2` onwards to `#1` onwards, like `C`'s `memcpy`.

```
\cs_new_protected:Npn \__regex_toks_memcpy:NNn #1#2#3
  {
    \prg_replicate:nn {#3}
      {
        \tex_toks:D #1 = \tex_toks:D #2
        \int_incr:N #1
        \int_incr:N #2
      }
  }
```

(*End of definition for* `\__regex_toks_memcpy:NNn`.)

`\__regex_toks_put_left:Ne`
`\__regex_toks_put_right:Ne`
`\__regex_toks_put_right:Nn`

During the building phase we wish to add `e`-expanded material to `\toks`, either to the left or to the right. The expansion is done "by hand" for optimization (these operations are used quite a lot). The `Nn` version of `\__regex_toks_put_right:Ne` is provided because it is more efficient than `e`-expanding with `\exp_not:n`.

```
\cs_if_exist:NTF \tex_etokspre:D
  { \cs_new_eq:NN \__regex_toks_put_left:Ne \tex_etokspre:D }
  {
    \cs_new_protected:Npn \__regex_toks_put_left:Ne #1#2
      { \tex_toks:D #1 = \tex_expanded:D {{ #2 \tex_the:D \tex_toks:D #1 }} }
  }
\cs_if_exist:NTF \tex_etoksapp:D
  { \cs_new_eq:NN \__regex_toks_put_right:Ne \tex_etoksapp:D }
  {
    \cs_new_protected:Npn \__regex_toks_put_right:Ne #1#2
      { \tex_toks:D #1 = \tex_expanded:D {{ \tex_the:D \tex_toks:D #1 #2 }} }
  }
\cs_if_exist:NTF \tex_toksapp:D
  { \cs_new_eq:NN \__regex_toks_put_right:Nn \tex_toksapp:D }
  {
    \cs_new_protected:Npn \__regex_toks_put_right:Nn #1#2
      { \tex_toks:D #1 = \exp_after:wN { \tex_the:D \tex_toks:D #1 #2 } }
  }
```

(*End of definition for* `\__regex_toks_put_left:Ne` *and* `\__regex_toks_put_right:Ne`.)

__regex_curr_cs_to_str: Expands to the string representation of the token (known to be a control sequence) at the current position `\l__regex_curr_pos_int`. It should only be used in e/x-expansion to avoid losing a leading space.

```
\cs_new:Npn \__regex_curr_cs_to_str:
  {
    \exp_after:wN \exp_after:wN \exp_after:wN \cs_to_str:N
    \l__regex_curr_token_tl
  }
```

(*End of definition for* `\__regex_curr_cs_to_str:`.)

__regex_intarray_item:NnF
__regex_intarray_item_aux:nNF

Item of intarray, with a default value.

```
\cs_new:Npn \__regex_intarray_item:NnF #1#2
  { \exp_args:No \__regex_intarray_item_aux:nNF { \tex_the:D \__regex_int_eval:w #2 } #1 }
\cs_new:Npn \__regex_intarray_item_aux:nNF #1#2
  {
    \if_int_compare:w #1 > \c_zero_int
      \exp_after:wN \use_ii:nnn
    \fi:
    \use_ii:nn { \__kernel_intarray_item:Nn #2 {#1} }
  }
```

(*End of definition for* `\__regex_intarray_item:NnF` *and* `\__regex_intarray_item_aux:nNF`.)

__regex_maplike_break: Analogous to `\tl_map_break:`, this correctly exits `\tl_map_inline:nn` and similar constructions and jumps to the matching `\prg_break_point:Nn \__regex_maplike_break: { }`.

```
\cs_new:Npn \__regex_maplike_break:
  { \prg_map_break:Nn \__regex_maplike_break: { } }
```

(*End of definition for* `\__regex_maplike_break:`.)

__regex_tl_odd_items:n
__regex_tl_even_items:n
__regex_tl_even_items_loop:nn

Map through a token list one pair at a time, leaving the odd-numbered or even-numbered items (the first item is numbered 1).

```
\cs_new:Npn \__regex_tl_odd_items:n #1 { \__regex_tl_even_items:n { ? #1 } }
\cs_new:Npn \__regex_tl_even_items:n #1
  {
    \__regex_tl_even_items_loop:nn #1 \q__regex_nil \q__regex_nil
    \prg_break_point:
  }
\cs_new:Npn \__regex_tl_even_items_loop:nn #1#2
  {
    \__regex_use_none_delimit_by_q_nil:w #2 \prg_break: \q__regex_nil
    { \exp_not:n {#2} }
    \__regex_tl_even_items_loop:nn
  }
```

(*End of definition for* `\__regex_tl_odd_items:n`, `\__regex_tl_even_items:n`, *and* `\__regex_tl_even_items_loop:nn`.)

46.2.1 Constants and variables

__regex_tmp:w

Temporary function used for various short-term purposes.

```
\cs_new:Npn \__regex_tmp:w { }
```

(*End of definition for* `\__regex_tmp:w`.)

\l__regex_internal_a_tl
\l__regex_internal_b_tl
\l__regex_internal_a_int
\l__regex_internal_b_int
\l__regex_internal_c_int
\l__regex_internal_bool
\l__regex_internal_seq
\g__regex_internal_tl

Temporary variables used for various purposes.

```
\tl_new:N   \l__regex_internal_a_tl
\tl_new:N   \l__regex_internal_b_tl
\int_new:N  \l__regex_internal_a_int
\int_new:N  \l__regex_internal_b_int
\int_new:N  \l__regex_internal_c_int
\bool_new:N \l__regex_internal_bool
\seq_new:N  \l__regex_internal_seq
\tl_new:N   \g__regex_internal_tl
```

(*End of definition for* `\l__regex_internal_a_tl` *and others.*)

\l__regex_build_tl

This temporary variable is specifically for use with the `tl_build` machinery.

```
\tl_new:N \l__regex_build_tl
```

(*End of definition for* `\l__regex_build_tl`.)

\c__regex_no_match_regex

This regular expression matches nothing, but is still a valid regular expression. We could use a failing assertion, but I went for an empty class. It is used as the initial value for regular expressions declared using `\regex_new:N`.

```
\tl_const:Nn \c__regex_no_match_regex
  {
    \__regex_branch:n
      { \__regex_class:NnnnN \c_true_bool { } { 1 } { 0 } \c_true_bool }
  }
```

(*End of definition for* `\c__regex_no_match_regex`.)

\l__regex_balance_int

During this phase, `\l__regex_balance_int` counts the balance of begin-group and end-group character tokens which appear before a given point in the token list. This variable is also used to keep track of the balance in the replacement text.

```
\int_new:N \l__regex_balance_int
```

(*End of definition for* `\l__regex_balance_int`.)

46.2.2 Testing characters

\c__regex_ascii_min_int
\c__regex_ascii_max_control_int
\c__regex_ascii_max_int

```
\int_const:Nn \c__regex_ascii_min_int { 0 }
\int_const:Nn \c__regex_ascii_max_control_int { 31 }
\int_const:Nn \c__regex_ascii_max_int { 127 }
```

(*End of definition for* `\c__regex_ascii_min_int`, `\c__regex_ascii_max_control_int`, *and* `\c__regex_ascii_max_int`.)

\c__regex_ascii_lower_int

```
\int_const:Nn \c__regex_ascii_lower_int { `a - `A }
```

(*End of definition for* `\c__regex_ascii_lower_int`.)

46.2.3 Internal auxiliaries

\q__regex_recursion_stop

Internal recursion quarks.

```
\quark_new:N \q__regex_recursion_stop
```

(*End of definition for* `\q__regex_recursion_stop`.)

\q__regex_nil

Internal quarks.

```
\quark_new:N \q__regex_nil
```

(*End of definition for* `\q__regex_nil`.)

egex_use_none_delimit_by_q_recursion_stop:w
_regex_use_i_delimit_by_q_recursion_stop:nw
__regex_use_none_delimit_by_q_nil:w

Functions to gobble up to a quark.

```
\cs_new:Npn \__regex_use_none_delimit_by_q_recursion_stop:w
  #1 \q__regex_recursion_stop { }
\cs_new:Npn \__regex_use_i_delimit_by_q_recursion_stop:nw
  #1 #2 \q__regex_recursion_stop {#1}
\cs_new:Npn \__regex_use_none_delimit_by_q_nil:w #1 \q__regex_nil { }
```

(*End of definition for* `\__regex_use_none_delimit_by_q_recursion_stop:w`, `\__regex_use_i_delimit_by_q_recursion_stop:nw`, *and* `\__regex_use_none_delimit_by_q_nil:w`.)

__regex_quark_if_nil_p:n
__regex_quark_if_nil:nTF

Branching quark conditional.

```
\__kernel_quark_new_conditional:Nn \__regex_quark_if_nil:N { F }
```

(*End of definition for* `\__regex_quark_if_nil:nTF`.)

__regex_break_point:TF
__regex_break_true:w

When testing whether a character of the query token list matches a given character class in the regular expression, we often have to test it against several ranges of characters, checking if any one of those matches. This is done with a structure like

> ⟨*test1*⟩ ... ⟨$test_n$⟩
> `\__regex_break_point:TF {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

If any of the tests succeeds, it calls `\__regex_break_true:w`, which cleans up and leaves ⟨*true code*⟩ in the input stream. Otherwise, `\__regex_break_point:TF` leaves the ⟨*false code*⟩ in the input stream.

```
\cs_new_protected:Npn \__regex_break_true:w
  #1 \__regex_break_point:TF #2 #3 {#2}
\cs_new_protected:Npn \__regex_break_point:TF #1 #2 { #2 }
```

(*End of definition for* `\__regex_break_point:TF` *and* `\__regex_break_true:w`.)

__regex_item_reverse:n

This function makes showing regular expressions easier, and lets us define `\D` in terms of `\d` for instance. There is a subtlety: the end of the query is marked by -2, and thus matches `\D` and other negated properties; this case is caught by another part of the code.

```
\cs_new_protected:Npn \__regex_item_reverse:n #1
  {
    #1
    \__regex_break_point:TF { } \__regex_break_true:w
  }
```

(*End of definition for* `\__regex_item_reverse:n`.)

__regex_item_caseful_equal:n
__regex_item_caseful_range:nn

Simple comparisons triggering `\__regex_break_true:w` when true.

```
\cs_new_protected:Npn \__regex_item_caseful_equal:n #1
  {
    \if_int_compare:w #1 = \l__regex_curr_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
\cs_new_protected:Npn \__regex_item_caseful_range:nn #1 #2
  {
    \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
      \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
      \fi:
    \fi:
  }
```

(*End of definition for* `\__regex_item_caseful_equal:n` *and* `\__regex_item_caseful_range:nn`*.*)

__regex_item_caseless_equal:n
__regex_item_caseless_range:nn

For caseless matching, we perform the test both on the `curr_char` and on the `case_changed_char`. Before doing the second set of tests, we make sure that `case_changed_char` has been computed.

```
\cs_new_protected:Npn \__regex_item_caseless_equal:n #1
  {
    \if_int_compare:w #1 = \l__regex_curr_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
    \__regex_maybe_compute_ccc:
    \if_int_compare:w #1 = \l__regex_case_changed_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
\cs_new_protected:Npn \__regex_item_caseless_range:nn #1 #2
  {
    \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
      \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
      \fi:
    \fi:
    \__regex_maybe_compute_ccc:
    \reverse_if:N \if_int_compare:w #1 > \l__regex_case_changed_char_int
      \reverse_if:N \if_int_compare:w #2 < \l__regex_case_changed_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
      \fi:
    \fi:
  }
```

(*End of definition for* `\__regex_item_caseless_equal:n` *and* `\__regex_item_caseless_range:nn`*.*)

__regex_compute_case_changed_char:

This function is called when `\l__regex_case_changed_char_int` has not yet been computed. If the current character code is in the range $[65, 90]$ (upper-case), then add 32, making it lowercase. If it is in the lower-case letter range $[97, 122]$, subtract 32.

```
\cs_new_protected:Npn \__regex_compute_case_changed_char:
  {
    \int_set_eq:NN \l__regex_case_changed_char_int \l__regex_curr_char_int
```

```
    \if_int_compare:w \l__regex_curr_char_int > `Z \exp_stop_f:
      \if_int_compare:w \l__regex_curr_char_int > `z \exp_stop_f: \else:
        \if_int_compare:w \l__regex_curr_char_int < `a \exp_stop_f: \else:
          \int_sub:Nn \l__regex_case_changed_char_int \c__regex_ascii_lower_int
        \fi:
      \fi:
    \else:
      \if_int_compare:w \l__regex_curr_char_int < `A \exp_stop_f: \else:
        \int_add:Nn \l__regex_case_changed_char_int \c__regex_ascii_lower_int
      \fi:
    \fi:
    \cs_set_eq:NN \__regex_maybe_compute_ccc: \prg_do_nothing:
  }
\cs_new_eq:NN \__regex_maybe_compute_ccc: \__regex_compute_case_changed_char:
```

(*End of definition for* `\__regex_compute_case_changed_char:`*.*)

`\__regex_item_equal:n`
`\__regex_item_range:nn`

Those must always be defined to expand to a `caseful` (default) or `caseless` version, and not be protected: they must expand when compiling, to hard-code which tests are caseless or caseful.

```
\cs_new_eq:NN \__regex_item_equal:n ?
\cs_new_eq:NN \__regex_item_range:nn ?
```

(*End of definition for* `\__regex_item_equal:n` *and* `\__regex_item_range:nn`*.*)

`\__regex_item_catcode:nT`
`\__regex_item_catcode_reverse:nT`
`\__regex_item_catcode:`

The argument is a sum of powers of 4 with exponents given by the allowed category codes (between 0 and 13). Dividing by a given power of 4 gives an odd result if and only if that category code is allowed. If the catcode does not match, then skip the character code tests which follow.

```
\cs_new_protected:Npn \__regex_item_catcode:
  {
    "
    \if_case:w \l__regex_curr_catcode_int
         1       \or: 4       \or: 10      \or: 40
    \or: 100     \or:         \or: 1000    \or: 4000
    \or: 10000   \or:         \or: 100000  \or: 400000
    \or: 1000000 \or: 4000000 \else: 1*0
    \fi:
  }
\prg_new_protected_conditional:Npnn \__regex_item_catcode:n #1 { T }
  {
    \if_int_odd:w \__regex_int_eval:w #1 / \__regex_item_catcode: \scan_stop:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new_protected:Npn \__regex_item_catcode_reverse:nT #1#2
  { \__regex_item_catcode:nT {#1} { \__regex_item_reverse:n {#2} } }
```

(*End of definition for* `\__regex_item_catcode:nT`*,* `\__regex_item_catcode_reverse:nT`*, and* `\__regex_item_catcode:`*.*)

__regex_item_exact:nn
__regex_item_exact_cs:n

This matches an exact ⟨*category*⟩-⟨*character code*⟩ pair, or an exact control sequence, more precisely one of several possible control sequences, separated by `\scan_stop:`.

```
\cs_new_protected:Npn \__regex_item_exact:nn #1#2
  {
    \if_int_compare:w #1 = \l__regex_curr_catcode_int
      \if_int_compare:w #2 = \l__regex_curr_char_int
        \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
      \fi:
    \fi:
  }
\cs_new_protected:Npn \__regex_item_exact_cs:n #1
  {
    \int_compare:nNnTF \l__regex_curr_catcode_int = \c_zero_int
      {
        \__kernel_tl_set:Nx \l__regex_internal_a_tl
          { \scan_stop: \__regex_curr_cs_to_str: \scan_stop: }
        \tl_if_in:noTF { \scan_stop: #1 \scan_stop: }
          \l__regex_internal_a_tl
          { \__regex_break_true:w } { }
      }
      { }
  }
```

(*End of definition for* `\__regex_item_exact:nn` *and* `\__regex_item_exact_cs:n`.)

__regex_item_cs:n

Match a control sequence (the argument is a compiled regex). First test the catcode of the current token to be zero. Then perform the matching test, and break if the csname indeed matches.

```
\cs_new_protected:Npn \__regex_item_cs:n #1
  {
    \int_compare:nNnT \l__regex_curr_catcode_int = \c_zero_int
      {
        \group_begin:
          \__regex_single_match:
          \__regex_disable_submatches:
          \__regex_build_for_cs:n {#1}
          \bool_set_eq:NN \l__regex_saved_success_bool
            \g__regex_success_bool
          \exp_args:Ne \__regex_match_cs:n { \__regex_curr_cs_to_str: }
          \if_meaning:w \c_true_bool \g__regex_success_bool
            \group_insert_after:N \__regex_break_true:w
          \fi:
          \bool_gset_eq:NN \g__regex_success_bool
            \l__regex_saved_success_bool
        \group_end:
      }
  }
```

(*End of definition for* `\__regex_item_cs:n`.)

46.2.4 Character property tests

__regex_prop_d:
__regex_prop_h:
__regex_prop_s:
__regex_prop_v:
__regex_prop_w:
__regex_prop_N:

Character property tests for `\d`, `\W`, *etc.* These character properties are not affected by the `(?i)` option. The characters recognized by each one are as follows: `\d=[0-9]`,

`\w=[0-9A-Z_a-z]`, `\s=[\␣\^^I\^^J\^^L\^^M]`, `\h=[\␣\^^I]`, `\v=[\^^J-\^^M]`, and the upper case counterparts match anything that the lower case does not match. The order in which the various tests appear is optimized for usual mostly lower case letter text.

```
\cs_new_protected:Npn \__regex_prop_d:
  { \__regex_item_caseful_range:nn { `0 } { `9 } }
\cs_new_protected:Npn \__regex_prop_h:
  {
    \__regex_item_caseful_equal:n { `\ }
    \__regex_item_caseful_equal:n { `\^^I }
  }
\cs_new_protected:Npn \__regex_prop_s:
  {
    \__regex_item_caseful_equal:n { `\ }
    \__regex_item_caseful_equal:n { `\^^I }
    \__regex_item_caseful_equal:n { `\^^J }
    \__regex_item_caseful_equal:n { `\^^L }
    \__regex_item_caseful_equal:n { `\^^M }
  }
\cs_new_protected:Npn \__regex_prop_v:
  { \__regex_item_caseful_range:nn { `\^^J } { `\^^M } } % lf, vtab, ff, cr
\cs_new_protected:Npn \__regex_prop_w:
  {
    \__regex_item_caseful_range:nn { `a } { `z }
    \__regex_item_caseful_range:nn { `A } { `Z }
    \__regex_item_caseful_range:nn { `0 } { `9 }
    \__regex_item_caseful_equal:n { `_ }
  }
\cs_new_protected:Npn \__regex_prop_N:
  {
    \__regex_item_reverse:n
      { \__regex_item_caseful_equal:n { `\^^J } }
  }
```

(*End of definition for* `\__regex_prop_d:` *and others.*)

`\__regex_posix_alnum:`
`\__regex_posix_alpha:`
`\__regex_posix_ascii:`
`\__regex_posix_blank:`
`\__regex_posix_cntrl:`
`\__regex_posix_digit:`
`\__regex_posix_graph:`
`\__regex_posix_lower:`
`\__regex_posix_print:`
`\__regex_posix_punct:`
`\__regex_posix_space:`
`\__regex_posix_upper:`
`\__regex_posix_word:`
`\__regex_posix_xdigit:`

POSIX properties. No surprise.

```
\cs_new_protected:Npn \__regex_posix_alnum:
  { \__regex_posix_alpha: \__regex_posix_digit: }
\cs_new_protected:Npn \__regex_posix_alpha:
  { \__regex_posix_lower: \__regex_posix_upper: }
\cs_new_protected:Npn \__regex_posix_ascii:
  {
    \__regex_item_caseful_range:nn
      \c__regex_ascii_min_int
      \c__regex_ascii_max_int
  }
\cs_new_eq:NN \__regex_posix_blank: \__regex_prop_h:
\cs_new_protected:Npn \__regex_posix_cntrl:
  {
    \__regex_item_caseful_range:nn
      \c__regex_ascii_min_int
      \c__regex_ascii_max_control_int
    \__regex_item_caseful_equal:n \c__regex_ascii_max_int
  }
```

```
\cs_new_eq:NN \__regex_posix_digit: \__regex_prop_d:
\cs_new_protected:Npn \__regex_posix_graph:
  { \__regex_item_caseful_range:nn { `! } { `\~ } }
\cs_new_protected:Npn \__regex_posix_lower:
  { \__regex_item_caseful_range:nn { `a } { `z } }
\cs_new_protected:Npn \__regex_posix_print:
  { \__regex_item_caseful_range:nn { `\  } { `\~ } }
\cs_new_protected:Npn \__regex_posix_punct:
  {
    \__regex_item_caseful_range:nn { `! } { `/ }
    \__regex_item_caseful_range:nn { `: } { `@ }
    \__regex_item_caseful_range:nn { `[ } { `` }
    \__regex_item_caseful_range:nn { `\{ } { `\~ }
  }
\cs_new_protected:Npn \__regex_posix_space:
  {
    \__regex_item_caseful_equal:n { `\  }
    \__regex_item_caseful_range:nn { `\^^I } { `\^^M }
  }
\cs_new_protected:Npn \__regex_posix_upper:
  { \__regex_item_caseful_range:nn { `A } { `Z } }
\cs_new_eq:NN \__regex_posix_word: \__regex_prop_w:
\cs_new_protected:Npn \__regex_posix_xdigit:
  {
    \__regex_posix_digit:
    \__regex_item_caseful_range:nn { `A } { `F }
    \__regex_item_caseful_range:nn { `a } { `f }
  }
```

(*End of definition for* `\__regex_posix_alnum:` *and others.*)

46.2.5 Simple character escape

Before actually parsing the regular expression or the replacement text, we go through them once, converting `\n` to the character 10, *etc.* In this pass, we also convert any special character (`*`, `?`, `{`, etc.) or escaped alphanumeric character into a marker indicating that this was a special sequence, and replace escaped special characters and non-escaped alphanumeric characters by markers indicating that those were "raw" characters. The rest of the code can then avoid caring about escaping issues (those can become quite complex to handle in combination with ranges in character classes).

Usage: `\__regex_escape_use:nnnn` ⟨*inline 1*⟩ ⟨*inline 2*⟩ ⟨*inline 3*⟩ `{`⟨*token list*⟩`}` The ⟨*token list*⟩ is converted to a string, then read from left to right, interpreting backslashes as escaping the next character. Unescaped characters are fed to the function ⟨*inline 1*⟩, and escaped characters are fed to the function ⟨*inline 2*⟩ within an `e`-expansion context (typically those functions perform some tests on their argument to decide how to output them). The escape sequences `\a`, `\e`, `\f`, `\n`, `\r`, `\t` and `\x` are recognized, and those are replaced by the corresponding character, then fed to ⟨*inline 3*⟩. The result is then left in the input stream. Spaces are ignored unless escaped.

The conversion is done within an `e`-expanding assignment.

`\__regex_escape_use:nnnn` The result is built in `\l__regex_internal_a_tl`, which is then left in the input stream. Tracing code is added as appropriate inside this token list. Go through `#4` once, applying

#1, #2, or #3 as relevant to each character (after de-escaping it).

```
\cs_new_protected:Npn \__regex_escape_use:nnnn #1#2#3#4
  {
    \group_begin:
      \tl_clear:N \l__regex_internal_a_tl
      \cs_set:Npn \__regex_escape_unescaped:N ##1 { #1 }
      \cs_set:Npn \__regex_escape_escaped:N ##1 { #2 }
      \cs_set:Npn \__regex_escape_raw:N ##1 { #3 }
      \__regex_standard_escapechar:
      \__kernel_tl_gset:Nx \g__regex_internal_tl
        { \__kernel_str_to_other_fast:n {#4} }
      \tl_put_right:Ne \l__regex_internal_a_tl
        {
          \exp_after:wN \__regex_escape_loop:N \g__regex_internal_tl
          \scan_stop: \prg_break_point:
        }
      \exp_after:wN
    \group_end:
    \l__regex_internal_a_tl
  }
```

(*End of definition for* `\__regex_escape_use:nnnn`.)

`\__regex_escape_loop:N`
`\__regex_escape_\:w`

`\__regex_escape_loop:N` reads one character: if it is special (space, backslash, or end-marker), perform the associated action, otherwise it is simply an unescaped character. After a backslash, the same is done, but unknown characters are "escaped".

```
\cs_new:Npn \__regex_escape_loop:N #1
  {
    \cs_if_exist_use:cF { __regex_escape_\token_to_str:N #1:w }
      { \__regex_escape_unescaped:N #1 }
    \__regex_escape_loop:N
  }
\cs_new:cpn { __regex_escape_ \c_backslash_str :w }
    \__regex_escape_loop:N #1
  {
    \cs_if_exist_use:cF { __regex_escape_/\token_to_str:N #1:w }
      { \__regex_escape_escaped:N #1 }
    \__regex_escape_loop:N
  }
```

(*End of definition for* `\__regex_escape_loop:N` *and* `\__regex_escape_\:w`.)

`\__regex_escape_unescaped:N`
`\__regex_escape_escaped:N`
`\__regex_escape_raw:N`

Those functions are never called before being given a new meaning, so their definitions here don't matter.

```
\cs_new_eq:NN \__regex_escape_unescaped:N ?
\cs_new_eq:NN \__regex_escape_escaped:N   ?
\cs_new_eq:NN \__regex_escape_raw:N       ?
```

(*End of definition for* `\__regex_escape_unescaped:N`, `\__regex_escape_escaped:N`, *and* `\__regex_escape_raw:N`.)

`\__regex_escape_\scan_stop::w`
`\__regex_escape_/\scan_stop::w`
`\__regex_escape_/a:w`
`\__regex_escape_/e:w`
`\__regex_escape_/f:w`
`\__regex_escape_/n:w`
`\__regex_escape_/r:w`
`\__regex_escape_/t:w`
`\__regex_escape_␣:w`

The loop is ended upon seeing the end-marker "`break`", with an error if the string ended in a backslash. Spaces are ignored, and `\a`, `\e`, `\f`, `\n`, `\r`, `\t` take their meaning here.

```
\cs_new_eq:cN { __regex_escape_ \iow_char:N\\scan_stop: :w } \prg_break:
```

```
\cs_new:cpn { __regex_escape_/ \iow_char:N\\scan_stop: :w }
  {
    \msg_expandable_error:nn { regex } { trailing-backslash }
    \prg_break:
  }
\cs_new:cpn { __regex_escape_~:w } { }
\cs_new:cpe { __regex_escape_/a:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^G }
\cs_new:cpe { __regex_escape_/t:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^I }
\cs_new:cpe { __regex_escape_/n:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^J }
\cs_new:cpe { __regex_escape_/f:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^L }
\cs_new:cpe { __regex_escape_/r:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^M }
\cs_new:cpe { __regex_escape_/e:w }
  { \exp_not:N \__regex_escape_raw:N \iow_char:N \^^[ }
```

(*End of definition for* `\__regex_escape_\scan_stop::w` *and others.*)

`\__regex_escape_/x:w`
`\__regex_escape_x_end:w`
`\__regex_escape_x_large:n`

When `\x` is encountered, `\__regex_escape_x_test:N` is responsible for grabbing some hexadecimal digits, and feeding the result to `\__regex_escape_x_end:w`. If the number is too big interrupt the assignment and produce an error, otherwise call `\__regex_escape_raw:N` on the corresponding character token.

```
\cs_new:cpn { __regex_escape_/x:w } \__regex_escape_loop:N
  {
    \exp_after:wN \__regex_escape_x_end:w
    \int_value:w "0 \__regex_escape_x_test:N
  }
\cs_new:Npn \__regex_escape_x_end:w #1 ;
  {
    \int_compare:nNnTF {#1} > \c_max_char_int
      {
        \msg_expandable_error:nnff { regex } { x-overflow }
          {#1} { \int_to_Hex:n {#1} }
      }
      {
        \exp_last_unbraced:Nf \__regex_escape_raw:N
          { \char_generate:nn {#1} { 12 } }
      }
  }
```

(*End of definition for* `\__regex_escape_/x:w`, `\__regex_escape_x_end:w`, *and* `\__regex_escape_x_large:n`.)

`\__regex_escape_x_test:N`
`\__regex_escape_x_testii:N`

Find out whether the first character is a left brace (allowing any number of hexadecimal digits), or not (allowing up to two hexadecimal digits). We need to check for the end-of-string marker. Eventually, call either `\__regex_escape_x_loop:N` or `\__regex_escape_x:N`.

```
\cs_new:Npn \__regex_escape_x_test:N #1
  {
    \if_meaning:w \scan_stop: #1
      \exp_after:wN \use_i:nnn \exp_after:wN ;
```

```
    \fi:
    \use:n
      {
        \if_charcode:w \c_space_token #1
          \exp_after:wN \__regex_escape_x_test:N
        \else:
          \exp_after:wN \__regex_escape_x_testii:N
          \exp_after:wN #1
        \fi:
      }
  }
\cs_new:Npn \__regex_escape_x_testii:N #1
  {
    \if_charcode:w \c_left_brace_str #1
      \exp_after:wN \__regex_escape_x_loop:N
    \else:
      \__regex_hexadecimal_use:NTF #1
        { \exp_after:wN \__regex_escape_x:N }
        { ; \exp_after:wN \__regex_escape_loop:N \exp_after:wN #1 }
    \fi:
  }
```

(*End of definition for* `\__regex_escape_x_test:N` *and* `\__regex_escape_x_testii:N`.)

`\__regex_escape_x:N` This looks for the second digit in the unbraced case.

```
\cs_new:Npn \__regex_escape_x:N #1
  {
    \if_meaning:w \scan_stop: #1
      \exp_after:wN \use_i:nnn \exp_after:wN ;
    \fi:
    \use:n
      {
        \__regex_hexadecimal_use:NTF #1
          { ; \__regex_escape_loop:N }
          { ; \__regex_escape_loop:N #1 }
      }
  }
```

(*End of definition for* `\__regex_escape_x:N`.)

`\__regex_escape_x_loop:N` `\__regex_escape_x_loop_error:` Grab hexadecimal digits, skip spaces, and at the end, check that there is a right brace, otherwise raise an error outside the assignment.

```
\cs_new:Npn \__regex_escape_x_loop:N #1
  {
    \if_meaning:w \scan_stop: #1
      \exp_after:wN \use_ii:nnn
    \fi:
    \use_ii:nn
      { ; \__regex_escape_x_loop_error:n { } {#1} }
      {
        \__regex_hexadecimal_use:NTF #1
          { \__regex_escape_x_loop:N }
          {
            \token_if_eq_charcode:NNTF \c_space_token #1
```

```
              { \__regex_escape_x_loop:N }
              {
                ;
                \exp_after:wN
                \token_if_eq_charcode:NNTF \c_right_brace_str #1
                  { \__regex_escape_loop:N }
                  { \__regex_escape_x_loop_error:n {#1} }
              }
          }
      }
  }
\cs_new:Npn \__regex_escape_x_loop_error:n #1
  {
    \msg_expandable_error:nnn { regex } { x-missing-rbrace } {#1}
    \__regex_escape_loop:N #1
  }
```

(*End of definition for* `\__regex_escape_x_loop:N` *and* `\__regex_escape_x_loop_error:`.)

`\__regex_hexadecimal_use:NTF`

TeX detects uppercase hexadecimal digits for us but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```
\cs_new:Npn \__regex_hexadecimal_use:NTF #1
  {
    \if_int_compare:w \c_one_int < "1 \token_to_str:N #1 \exp_stop_f:
      #1
    \else:
      \if_case:w
        \__regex_int_eval:w \exp_after:wN ` \token_to_str:N #1 - `a \scan_stop:
           A
      \or: B
      \or: C
      \or: D
      \or: E
      \or: F
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN \use_iii:nnn
      \fi:
    \fi:
    \use_i:nn
  }
```

(*End of definition for* `\__regex_hexadecimal_use:NTF`.)

`\__regex_char_if_alphanumeric:NTF`
`\__regex_char_if_special:NTF`

These two tests are used in the first pass when parsing a regular expression. That pass is responsible for finding escaped and non-escaped characters, and recognizing which ones have special meanings and which should be interpreted as "raw" characters. Namely,

- alphanumerics are "raw" if they are not escaped, and may have a special meaning when escaped;
- non-alphanumeric printable ascii characters are "raw" if they are escaped, and may have a special meaning when not escaped;
- characters other than printable ascii are always "raw".

The code is ugly, and highly based on magic numbers and the ascii codes of characters. This is mostly unavoidable for performance reasons. Maybe the tests can be optimized a little bit more. Here, “alphanumeric” means `0`–`9`, `A`–`Z`, `a`–`z`; “special” character means non-alphanumeric but printable ascii, from space (hex `20`) to `del` (hex `7E`).

```
\prg_new_conditional:Npnn \__regex_char_if_special:N #1 { TF }
  {
    \if:w
        T
        \if_int_compare:w ‘#1 > ‘Z \exp_stop_f:
          \if_int_compare:w ‘#1 > ‘z \exp_stop_f:
            \if_int_compare:w ‘#1 < \c__regex_ascii_max_int
              \else: F \fi:
          \else:
            \if_int_compare:w ‘#1 < ‘a \exp_stop_f:
              \else: F \fi:
          \fi:
        \else:
          \if_int_compare:w ‘#1 > ‘9 \exp_stop_f:
            \if_int_compare:w ‘#1 < ‘A \exp_stop_f:
              \else: F \fi:
          \else:
            \if_int_compare:w ‘#1 < ‘0 \exp_stop_f:
              \if_int_compare:w ‘#1 < ‘\ \exp_stop_f:
                F \fi:
            \else: F \fi:
          \fi:
        \fi:
        T
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_new_conditional:Npnn \__regex_char_if_alphanumeric:N #1 { TF }
  {
    \if:w
        T
        \if_int_compare:w ‘#1 > ‘Z \exp_stop_f:
          \if_int_compare:w ‘#1 > ‘z \exp_stop_f:
            F
          \else:
            \if_int_compare:w ‘#1 < ‘a \exp_stop_f:
              F \fi:
          \fi:
        \else:
          \if_int_compare:w ‘#1 > ‘9 \exp_stop_f:
            \if_int_compare:w ‘#1 < ‘A \exp_stop_f:
              F \fi:
          \else:
            \if_int_compare:w ‘#1 < ‘0 \exp_stop_f:
              F \fi:
          \fi:
        \fi:
        T
```

```
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\__regex_char_if_alphanumeric:NTF` *and* `\__regex_char_if_special:NTF`.)

46.3 Compiling

A regular expression starts its life as a string of characters. In this section, we convert it to internal instructions, resulting in a "compiled" regular expression. This compiled expression is then turned into states of an automaton in the building phase. Compiled regular expressions consist of the following:

- `\__regex_class:NnnnN` ⟨**boolean**⟩ {⟨*tests*⟩} {⟨*min*⟩} {⟨*more*⟩} ⟨**laziness**⟩
- `\__regex_group:nnnN` {⟨*branches*⟩} {⟨*min*⟩} {⟨*more*⟩} ⟨**laziness**⟩, also `\__regex_group_no_capture:nnnN` and `\__regex_group_resetting:nnnN` with the same syntax.
- `\__regex_branch:n` {⟨*contents*⟩}
- `\__regex_command_K:`
- `\__regex_assertion:Nn` ⟨**boolean**⟩ {⟨*assertion test*⟩}, where the ⟨**assertion test**⟩ is `\__regex_b_test:` or `\__regex_Z_test:` or `\__regex_A_test:` or `\__regex_G_test:`

Tests can be the following:

- `\__regex_item_caseful_equal:n` {⟨*char code*⟩}
- `\__regex_item_caseless_equal:n` {⟨*char code*⟩}
- `\__regex_item_caseful_range:nn` {⟨*min*⟩} {⟨*max*⟩}
- `\__regex_item_caseless_range:nn` {⟨*min*⟩} {⟨*max*⟩}
- `\__regex_item_catcode:nT` {⟨*catcode bitmap*⟩} {⟨*tests*⟩}
- `\__regex_item_catcode_reverse:nT` {⟨*catcode bitmap*⟩} {⟨*tests*⟩}
- `\__regex_item_reverse:n` {⟨*tests*⟩}
- `\__regex_item_exact:nn` {⟨*catcode*⟩} {⟨*char code*⟩}
- `\__regex_item_exact_cs:n` {⟨*csnames*⟩}, more precisely given as ⟨**csname**⟩ `\scan_stop:` ⟨**csname**⟩ `\scan_stop:` ⟨**csname**⟩ and so on in a brace group.
- `\__regex_item_cs:n` {⟨*compiled regex*⟩}

46.3.1 Variables used when compiling

\l__regex_group_level_int

We make sure to open the same number of groups as we close.

```
\int_new:N \l__regex_group_level_int
```

(*End of definition for* `\l__regex_group_level_int`*.*)

\l__regex_mode_int
\c__regex_cs_in_class_mode_int
\c__regex_cs_mode_int
\c__regex_outer_mode_int
\c__regex_catcode_mode_int
\c__regex_class_mode_int
\c__regex_catcode_in_class_mode_int

While compiling, ten modes are recognized, labelled -63, -23, -6, -2, 0, 2, 3, 6, 23, 63. See section 46.3.3. We only define some of these as constants.

```
\int_new:N \l__regex_mode_int
\int_const:Nn \c__regex_cs_in_class_mode_int { -6 }
\int_const:Nn \c__regex_cs_mode_int { -2 }
\int_const:Nn \c__regex_outer_mode_int { 0 }
\int_const:Nn \c__regex_catcode_mode_int { 2 }
\int_const:Nn \c__regex_class_mode_int { 3 }
\int_const:Nn \c__regex_catcode_in_class_mode_int { 6 }
```

(*End of definition for* `\l__regex_mode_int` *and others.*)

\l__regex_catcodes_int
\l__regex_default_catcodes_int
\l__regex_catcodes_bool

We wish to allow constructions such as `\c[^BE](..\cL[a-z]..)`, where the outer catcode test applies to the whole group, but is superseded by the inner catcode test. For this to work, we need to keep track of lists of allowed category codes: `\l__regex_catcodes_int` and `\l__regex_default_catcodes_int` are bitmaps, sums of 4^c, for all allowed catcodes c. The latter is local to each capturing group, and we reset `\l__regex_catcodes_int` to that value after each character or class, changing it only when encountering a `\c` escape. The boolean records whether the list of categories of a catcode test has to be inverted: compare `\c[^BE]` and `\c[BE]`.

```
\int_new:N \l__regex_catcodes_int
\int_new:N \l__regex_default_catcodes_int
\bool_new:N \l__regex_catcodes_bool
```

(*End of definition for* `\l__regex_catcodes_int`*,* `\l__regex_default_catcodes_int`*, and* `\l__regex_catcodes_bool`*.*)

\c__regex_catcode_C_int
\c__regex_catcode_B_int
\c__regex_catcode_E_int
\c__regex_catcode_M_int
\c__regex_catcode_T_int
\c__regex_catcode_P_int
\c__regex_catcode_U_int
\c__regex_catcode_D_int
\c__regex_catcode_S_int
\c__regex_catcode_L_int
\c__regex_catcode_O_int
\c__regex_catcode_A_int
\c__regex_all_catcodes_int

Constants: 4^c for each category, and the sum of all powers of 4.

```
\int_const:Nn \c__regex_catcode_C_int { "1 }
\int_const:Nn \c__regex_catcode_B_int { "4 }
\int_const:Nn \c__regex_catcode_E_int { "10 }
\int_const:Nn \c__regex_catcode_M_int { "40 }
\int_const:Nn \c__regex_catcode_T_int { "100 }
\int_const:Nn \c__regex_catcode_P_int { "1000 }
\int_const:Nn \c__regex_catcode_U_int { "4000 }
\int_const:Nn \c__regex_catcode_D_int { "10000 }
\int_const:Nn \c__regex_catcode_S_int { "100000 }
\int_const:Nn \c__regex_catcode_L_int { "400000 }
\int_const:Nn \c__regex_catcode_O_int { "1000000 }
\int_const:Nn \c__regex_catcode_A_int { "4000000 }
\int_const:Nn \c__regex_all_catcodes_int { "5515155 }
```

(*End of definition for* `\c__regex_catcode_C_int` *and others.*)

\l__regex_internal_regex

The compilation step stores its result in this variable.

```
\cs_new_eq:NN \l__regex_internal_regex \c__regex_no_match_regex
```

(*End of definition for* `\l__regex_internal_regex`*.*)

\l__regex_show_prefix_seq

This sequence holds the prefix that makes up the line displayed to the user. The various items must be removed from the right, which is tricky with a token list, hence we use a sequence.

```
\seq_new:N \l__regex_show_prefix_seq
```

(*End of definition for* `\l__regex_show_prefix_seq`*.*)

\l__regex_show_lines_int

A hack. To know whether a given class has a single item in it or not, we count the number of lines when showing the class.

```
\int_new:N \l__regex_show_lines_int
```

(*End of definition for* `\l__regex_show_lines_int`*.*)

46.3.2 Generic helpers used when compiling

__regex_two_if_eq:NNNNTF

Used to compare pairs of things like `\__regex_compile_special:N ?` together. It's often inconvenient to get the catcodes of the character to match so we just compare the character code. Besides, the expanding behaviour of `\if:w` is very useful as that means we can use `\c_left_brace_str` and the like.

```
\cs_new:Npn \__regex_two_if_eq:NNNNTF #1#2#3#4
  {
    \if_meaning:w #1 #3
      \if:w #2 #4
        \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nnn
      \fi:
    \fi:
    \use_ii:nn
  }
```

(*End of definition for* `\__regex_two_if_eq:NNNNTF`*.*)

__regex_get_digits:NTFw
__regex_get_digits_loop:w

If followed by some raw digits, collect them one by one in the integer variable `#1`, and take the `true` branch. Otherwise, take the `false` branch.

```
\cs_new_protected:Npn \__regex_get_digits:NTFw #1#2#3#4#5
  {
    \__regex_if_raw_digit:NNTF #4 #5
      { #1 = #5 \__regex_get_digits_loop:nw {#2} }
      { #3 #4 #5 }
  }
\cs_new:Npn \__regex_get_digits_loop:nw #1#2#3
  {
    \__regex_if_raw_digit:NNTF #2 #3
      { #3 \__regex_get_digits_loop:nw {#1} }
      { \scan_stop: #1 #2 #3 }
  }
```

(*End of definition for* `\__regex_get_digits:NTFw` *and* `\__regex_get_digits_loop:w`*.*)

__regex_if_raw_digit:NNTF

Test used when grabbing digits for the `{m,n}` quantifier. It only accepts non-escaped digits.

```
\cs_new:Npn \__regex_if_raw_digit:NNTF #1#2
  {
    \if_meaning:w \__regex_compile_raw:N #1
      \if_int_compare:w \c_one_int < 1 #2 \exp_stop_f:
```

```
      \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nnn
    \fi:
  \fi:
  \use_ii:nn
}
```

(*End of definition for* `\__regex_if_raw_digit:NNTF`.)

46.3.3 Mode

When compiling the NFA corresponding to a given regex string, we can be in ten distinct modes, which we label by some magic numbers:

-6 `[\c{...}]` control sequence in a class,

-2 `\c{...}` control sequence,

0 ... outer,

2 `\c...` catcode test,

6 `[\c...]` catcode test in a class,

-63 `[\c{[...]}]` class inside mode -6,

-23 `\c{[...]}` class inside mode -2,

3 `[...]` class inside mode 0,

23 `\c[...]` class inside mode 2,

63 `[\c[...]]` class inside mode 6.

This list is exhaustive, because `\c` escape sequences cannot be nested, and character classes cannot be nested directly. The choice of numbers is such as to optimize the most useful tests, and make transitions from one mode to another as simple as possible.

- Even modes mean that we are not directly in a character class. In this case, a left bracket appends 3 to the mode. In a character class, a right bracket changes the mode as $m \to (m-15)/13$, truncated.
- Grouping, assertion, and anchors are allowed in non-positive even modes (0, -2, -6), and do not change the mode. Otherwise, they trigger an error.
- A left bracket is special in even modes, appending 3 to the mode; in those modes, quantifiers and the dot are recognized, and the right bracket is normal. In odd modes (within classes), the left bracket is normal, but the right bracket ends the class, changing the mode from m to $(m-15)/13$, truncated; also, ranges are recognized.
- In non-negative modes, left and right braces are normal. In negative modes, however, left braces trigger a warning; right braces end the control sequence, going from -2 to 0 or -6 to 3, with error recovery for odd modes.
- Properties (such as the `\d` character class) can appear in any mode.

__regex_if_in_class:TF Test whether we are directly in a character class (at the innermost level of nesting). There, many escape sequences are not recognized, and special characters are normal. Also, for every raw character, we must look ahead for a possible raw dash.

```
\prg_new_conditional:Npnn \__regex_if_in_class: { TF }
  {
    \if_int_odd:w \l__regex_mode_int
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\__regex_if_in_class:TF`*.*)

__regex_if_in_cs:TF Right braces are special only directly inside control sequences (at the inner-most level of nesting, not counting groups).

```
\cs_new:Npn \__regex_if_in_cs:TF
  {
    \if_int_odd:w \l__regex_mode_int
    \else:
      \if_int_compare:w \l__regex_mode_int < \c__regex_outer_mode_int
        \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nnn
      \fi:
    \fi:
    \use_ii:nn
  }
```

(*End of definition for* `\__regex_if_in_cs:TF`*.*)

__regex_if_in_class_or_catcode:TF Assertions are only allowed in modes 0, -2, and -6, *i.e.*, even, non-positive modes.

```
\cs_new:Npn \__regex_if_in_class_or_catcode:TF
  {
    \if_int_odd:w \l__regex_mode_int
    \else:
      \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN \use_iii:nnn
      \fi:
    \fi:
    \use_i:nn
  }
```

(*End of definition for* `\__regex_if_in_class_or_catcode:TF`*.*)

__regex_if_within_catcode:TF This test takes the true branch if we are in a catcode test, either immediately following it (modes 2 and 6) or in a class on which it applies (modes 23 and 63). This is used to tweak how left brackets behave in modes 2 and 6.

```
\prg_new_conditional:Npnn \__regex_if_within_catcode: { TF }
  {
    \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\__regex_if_within_catcode:TF`.)

`\__regex_chk_c_allowed:T` The `\c` escape sequence is only allowed in modes 0 and 3, *i.e.*, not within any other `\c` escape sequence.

```
\cs_new_protected:Npn \__regex_chk_c_allowed:T
  {
    \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
    \else:
      \if_int_compare:w \l__regex_mode_int = \c__regex_class_mode_int
      \else:
        \msg_error:nn { regex } { c-bad-mode }
        \exp_after:wN \use_i:nnn
      \fi:
    \fi:
    \use:n
  }
```

(*End of definition for* `\__regex_chk_c_allowed:T`.)

`\__regex_mode_quit_c:` This function changes the mode as it is needed just after a catcode test.

```
\cs_new_protected:Npn \__regex_mode_quit_c:
  {
    \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
      \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
    \else:
      \if_int_compare:w \l__regex_mode_int =
        \c__regex_catcode_in_class_mode_int
        \int_set_eq:NN \l__regex_mode_int \c__regex_class_mode_int
      \fi:
    \fi:
  }
```

(*End of definition for* `\__regex_mode_quit_c:`.)

46.3.4 Framework

`\__regex_compile:w`
`\__regex_compile_end:`

Used when compiling a user regex or a regex for the `\c{...}` escape sequence within another regex. Start building a token list within a group (with e-expansion at the outset), and set a few variables (group level, catcodes), then start the first branch. At the end, make sure there are no dangling classes nor groups, close the last branch: we are done building `\l__regex_internal_regex`.

```
\cs_new_protected:Npn \__regex_compile:w
  {
    \group_begin:
      \tl_build_begin:N \l__regex_build_tl
      \int_zero:N \l__regex_group_level_int
      \int_set_eq:NN \l__regex_default_catcodes_int
        \c__regex_all_catcodes_int
      \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
      \cs_set:Npn \__regex_item_equal:n  { \__regex_item_caseful_equal:n }
      \cs_set:Npn \__regex_item_range:nn { \__regex_item_caseful_range:nn }
      \tl_build_put_right:Nn \l__regex_build_tl
        { \__regex_branch:n { \if_false: } \fi: }
  }
```

```
\cs_new_protected:Npn \__regex_compile_end:
  {
      \__regex_if_in_class:TF
        {
          \msg_error:nn { regex } { missing-rbrack }
          \use:c { __regex_compile_]: }
          \prg_do_nothing: \prg_do_nothing:
        }
        { }
      \if_int_compare:w \l__regex_group_level_int > \c_zero_int
        \msg_error:nne { regex } { missing-rparen }
          { \int_use:N \l__regex_group_level_int }
        \prg_replicate:nn
          \l__regex_group_level_int
          {
              \tl_build_put_right:Nn \l__regex_build_tl
                {
                  \if_false: { \fi: }
                  \if_false: { \fi: } { 1 } { 0 } \c_true_bool
                }
              \tl_build_end:N \l__regex_build_tl
              \exp_args:NNNo
            \group_end:
            \tl_build_put_right:Nn \l__regex_build_tl
              { \l__regex_build_tl }
          }
      \fi:
      \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
      \tl_build_end:N \l__regex_build_tl
      \exp_args:NNNe
    \group_end:
    \tl_set:Nn \l__regex_internal_regex { \l__regex_build_tl }
  }
```

(*End of definition for* `\__regex_compile:w` *and* `\__regex_compile_end:`.)

`\__regex_compile:n` The compilation is done between `\__regex_compile:w` and `\__regex_compile_end:`, starting in mode 0. Then `\__regex_escape_use:nnnn` distinguishes special characters, escaped alphanumerics, and raw characters, interpreting `\a`, `\x` and other sequences. The 4 trailing `\prg_do_nothing:` are needed because some functions defined later look up to 4 tokens ahead. Before ending, make sure that any `\c{...}` is properly closed. No need to check that brackets are closed properly since `\__regex_compile_end:` does that. However, catch the case of a trailing `\cL` construction.

```
\cs_new_protected:Npn \__regex_compile:n #1
  {
    \__regex_compile:w
      \__regex_standard_escapechar:
      \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
      \__regex_escape_use:nnnn
        {
          \__regex_char_if_special:NTF ##1
            \__regex_compile_special:N \__regex_compile_raw:N ##1
        }
        {
```

```
          \__regex_char_if_alphanumeric:NTF ##1
            \__regex_compile_escaped:N \__regex_compile_raw:N ##1
        }
        { \__regex_compile_raw:N ##1 }
        { #1 }
      \prg_do_nothing: \prg_do_nothing:
      \prg_do_nothing: \prg_do_nothing:
      \int_compare:nNnT \l__regex_mode_int = \c__regex_catcode_mode_int
        { \msg_error:nn { regex } { c-trailing } }
      \int_compare:nNnT \l__regex_mode_int < \c__regex_outer_mode_int
        {
          \msg_error:nn { regex } { c-missing-rbrace }
          \__regex_compile_end_cs:
          \prg_do_nothing: \prg_do_nothing:
          \prg_do_nothing: \prg_do_nothing:
        }
    \__regex_compile_end:
  }
```

(*End of definition for* `\__regex_compile:n`*.*)

`\__regex_compile_use:n`

Use a regex, regardless of whether it is given as a string (in which case we need to compile) or as a regex variable. This is used for `\regex_match_case:nn` and related functions to allow a mixture of explicit regex and regex variables.

```
\cs_new_protected:Npn \__regex_compile_use:n #1
  {
    \tl_if_single_token:nT {#1}
      {
        \exp_after:wN \__regex_compile_use_aux:w
        \token_to_meaning:N #1 ~ \q__regex_nil
      }
    \__regex_compile:n {#1} \l__regex_internal_regex
  }
\cs_new_protected:Npn \__regex_compile_use_aux:w #1 ~ #2 \q__regex_nil
  {
    \str_if_eq:nnT { #1 ~ } { macro:->\__regex_branch:n }
      { \use_ii:nnn }
  }
```

(*End of definition for* `\__regex_compile_use:n`*.*)

`\__regex_compile_escaped:N`
`\__regex_compile_special:N`

If the special character or escaped alphanumeric has a particular meaning in regexes, the corresponding function is used. Otherwise, it is interpreted as a raw character. We distinguish special characters from escaped alphanumeric characters because they behave differently when appearing as an end-point of a range.

```
\cs_new_protected:Npn \__regex_compile_special:N #1
  {
    \cs_if_exist_use:cF { __regex_compile_#1: }
      { \__regex_compile_raw:N #1 }
  }
\cs_new_protected:Npn \__regex_compile_escaped:N #1
  {
    \cs_if_exist_use:cF { __regex_compile_/#1: }
      { \__regex_compile_raw:N #1 }
  }
```

(*End of definition for* `\__regex_compile_escaped:N` *and* `\__regex_compile_special:N`.)

`\__regex_compile_one:n`

This is used after finding one "test", such as `\d`, or a raw character. If that followed a catcode test (*e.g.*, `\cL`), then restore the mode. If we are not in a class, then the test is "standalone", and we need to add `\__regex_class:NnnnN` and search for quantifiers. In any case, insert the test, possibly together with a catcode test if appropriate.

```
\cs_new_protected:Npn \__regex_compile_one:n #1
  {
    \__regex_mode_quit_c:
    \__regex_if_in_class:TF { }
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
      }
    \tl_build_put_right:Ne \l__regex_build_tl
      {
        \if_int_compare:w \l__regex_catcodes_int <
          \c__regex_all_catcodes_int
          \__regex_item_catcode:nT { \int_use:N \l__regex_catcodes_int }
            { \exp_not:N \exp_not:n {#1} }
        \else:
          \exp_not:N \exp_not:n {#1}
        \fi:
      }
    \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
    \__regex_if_in_class:TF { } { \__regex_compile_quantifier:w }
  }
```

(*End of definition for* `\__regex_compile_one:n`.)

`\__regex_compile_abort_tokens:n`
`\__regex_compile_abort_tokens:e`

This function places the collected tokens back in the input stream, each as a raw character. Spaces are not preserved.

```
\cs_new_protected:Npn \__regex_compile_abort_tokens:n #1
  {
    \use:e
      {
        \exp_args:No \tl_map_function:nN { \tl_to_str:n {#1} }
          \__regex_compile_raw:N
      }
  }
\cs_generate_variant:Nn \__regex_compile_abort_tokens:n { e }
```

(*End of definition for* `\__regex_compile_abort_tokens:n`.)

46.3.5 Quantifiers

`\__regex_compile_if_quantifier:TFw`

This looks ahead and checks whether there are any quantifier (special character equal to either of `?+*{`). This is useful for the `\u` and `\ur` escape sequences.

```
\cs_new_protected:Npn \__regex_compile_if_quantifier:TFw #1#2#3#4
  {
    \token_if_eq_meaning:NNTF #3 \__regex_compile_special:N
      { \cs_if_exist:cTF { __regex_compile_quantifier_#4:w } }
      { \use_ii:nn }
    {#1} {#2} #3 #4
```

```
  }
```

(*End of definition for* `\__regex_compile_if_quantifier:TFw`*.*)

`\__regex_compile_quantifier:w`

This looks ahead and finds any quantifier (special character equal to either of `?+*{`).

```
\cs_new_protected:Npn \__regex_compile_quantifier:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
      {
        \cs_if_exist_use:cF { __regex_compile_quantifier_#2:w }
          { \__regex_compile_quantifier_none: #1 #2 }
      }
      { \__regex_compile_quantifier_none: #1 #2 }
  }
```

(*End of definition for* `\__regex_compile_quantifier:w`*.*)

`\__regex_compile_quantifier_none:`
`\__regex_compile_quantifier_abort:eNN`

Those functions are called whenever there is no quantifier, or a braced construction is invalid (equivalent to no quantifier, and whatever characters were grabbed are left raw).

```
\cs_new_protected:Npn \__regex_compile_quantifier_none:
  {
    \tl_build_put_right:Nn \l__regex_build_tl
      { \if_false: { \fi: } { 1 } { 0 } \c_false_bool }
  }
\cs_new_protected:Npn \__regex_compile_quantifier_abort:eNN #1#2#3
  {
    \__regex_compile_quantifier_none:
    \msg_warning:nnee { regex } { invalid-quantifier } {#1} {#3}
    \__regex_compile_abort_tokens:e {#1}
    #2 #3
  }
```

(*End of definition for* `\__regex_compile_quantifier_none:` *and* `\__regex_compile_quantifier_abort:eNN`*.*)

`\__regex_compile_quantifier_laziness:nnNN`

Once the "main" quantifier (`?`, `*`, `+` or a braced construction) is found, we check whether it is lazy (followed by a question mark). We then add to the compiled regex a closing brace (ending `\__regex_class:NnnnN` and friends), the start-point of the range, its end-point, and a boolean, `true` for lazy and `false` for greedy operators.

```
\cs_new_protected:Npn \__regex_compile_quantifier_laziness:nnNN #1#2#3#4
  {
    \__regex_two_if_eq:NNNNTF #3 #4 \__regex_compile_special:N ?
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          { \if_false: { \fi: } { #1 } { #2 } \c_true_bool }
      }
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          { \if_false: { \fi: } { #1 } { #2 } \c_false_bool }
        #3 #4
      }
  }
```

(*End of definition for* `\__regex_compile_quantifier_laziness:nnNN`*.*)

__regex_compile_quantifier_?:w
__regex_compile_quantifier_*:w
__regex_compile_quantifier_+:w

For each "basic" quantifier, `?`, `*`, `+`, feed the correct arguments to `\__regex_compile_quantifier_laziness:nnNN`, -1 means that there is no upper bound on the number of repetitions.

```
\cs_new_protected:cpn { __regex_compile_quantifier_?:w }
  { \__regex_compile_quantifier_laziness:nnNN { 0 } { 1 } }
\cs_new_protected:cpn { __regex_compile_quantifier_*:w }
  { \__regex_compile_quantifier_laziness:nnNN { 0 } { -1 } }
\cs_new_protected:cpn { __regex_compile_quantifier_+:w }
  { \__regex_compile_quantifier_laziness:nnNN { 1 } { -1 } }
```

(*End of definition for* `\__regex_compile_quantifier_?:w`*,* `\__regex_compile_quantifier_*:w`*, and* `\__regex_compile_quantifier_+:w`*.*)

__regex_compile_quantifier_{:w
__regex_compile_quantifier_braced_auxi:w
__regex_compile_quantifier_braced_auxii:w
__regex_compile_quantifier_braced_auxiii:w

Three possible syntaxes: `{`⟨*int*⟩`}`, `{`⟨*int*⟩`,}`, or `{`⟨*int*⟩`,`⟨*int*⟩`}`. Any other syntax causes us to abort and put whatever we collected back in the input stream, as `raw` characters, including the opening brace. Grab a number into `\l__regex_internal_a_int`. If the number is followed by a right brace, the range is $[a, a]$. If followed by a comma, grab one more number, and call the `_ii` or `_iii` auxiliary. Those auxiliaries check for a closing brace, leading to the range $[a, \infty]$ or $[a, b]$, encoded as $\{a\}\{-1\}$ and $\{a\}\{b-a\}$.

```
\cs_new_protected:cpn { __regex_compile_quantifier_ \c_left_brace_str :w }
  {
    \__regex_get_digits:NTFw \l__regex_internal_a_int
      { \__regex_compile_quantifier_braced_auxi:w }
      { \__regex_compile_quantifier_abort:eNN { \c_left_brace_str } }
  }
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxi:w #1#2
  {
    \str_case_e:nnF { #1 #2 }
      {
        { \__regex_compile_special:N \c_right_brace_str }
          {
            \exp_args:No \__regex_compile_quantifier_laziness:nnNN
              { \int_use:N \l__regex_internal_a_int } 0
          }
        { \__regex_compile_special:N , }
          {
            \__regex_get_digits:NTFw \l__regex_internal_b_int
              { \__regex_compile_quantifier_braced_auxiii:w }
              { \__regex_compile_quantifier_braced_auxii:w }
          }
      }
      {
        \__regex_compile_quantifier_abort:eNN
          { \c_left_brace_str \int_use:N \l__regex_internal_a_int }
        #1 #2
      }
  }
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxii:w #1#2
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N \c_right_brace_str
      {
        \exp_args:No \__regex_compile_quantifier_laziness:nnNN
          { \int_use:N \l__regex_internal_a_int } { -1 }
      }
```

```
      {
        \__regex_compile_quantifier_abort:eNN
          { \c_left_brace_str \int_use:N \l__regex_internal_a_int , }
        #1 #2
      }
  }
\cs_new_protected:Npn \__regex_compile_quantifier_braced_auxiii:w #1#2
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N \c_right_brace_str
      {
        \if_int_compare:w \l__regex_internal_a_int >
          \l__regex_internal_b_int
          \msg_error:nnee { regex } { backwards-quantifier }
            { \int_use:N \l__regex_internal_a_int }
            { \int_use:N \l__regex_internal_b_int }
          \int_zero:N \l__regex_internal_b_int
        \else:
          \int_sub:Nn \l__regex_internal_b_int \l__regex_internal_a_int
        \fi:
        \exp_args:Noo \__regex_compile_quantifier_laziness:nnNN
          { \int_use:N \l__regex_internal_a_int }
          { \int_use:N \l__regex_internal_b_int }
      }
      {
        \__regex_compile_quantifier_abort:eNN
          {
            \c_left_brace_str
            \int_use:N \l__regex_internal_a_int ,
            \int_use:N \l__regex_internal_b_int
          }
        #1 #2
      }
  }
```

(*End of definition for* `\__regex_compile_quantifier_{:w` *and others.*)

46.3.6 Raw characters

`\__regex_compile_raw_error:N` Within character classes, and following catcode tests, some escaped alphanumeric sequences such as `\b` do not have any meaning. They are replaced by a raw character, after spitting out an error.

```
\cs_new_protected:Npn \__regex_compile_raw_error:N #1
  {
    \msg_error:nne { regex } { bad-escape } {#1}
    \__regex_compile_raw:N #1
  }
```

(*End of definition for* `\__regex_compile_raw_error:N`.)

`\__regex_compile_raw:N` If we are in a character class and the next character is an unescaped dash, this denotes a range. Otherwise, the current character `#1` matches itself.

```
\cs_new_protected:Npn \__regex_compile_raw:N #1#2#3
  {
    \__regex_if_in_class:TF
```

```
      {
        \__regex_two_if_eq:NNNNTF #2 #3 \__regex_compile_special:N -
          { \__regex_compile_range:Nw #1 }
          {
            \__regex_compile_one:n
              { \__regex_item_equal:n { \int_value:w `#1 } }
            #2 #3
          }
      }
      {
        \__regex_compile_one:n
          { \__regex_item_equal:n { \int_value:w `#1 } }
        #2 #3
      }
  }
```

(*End of definition for* `\__regex_compile_raw:N`.)

`\__regex_compile_range:Nw`
`\__regex_if_end_range:NNTF`

We have just read a raw character followed by a dash; this should be followed by an end-point for the range. Valid end-points are: any raw character; any special character, except a right bracket. In particular, escaped characters are forbidden.

```
\cs_new_protected:Npn \__regex_if_end_range:NNTF #1#2
  {
    \if_meaning:w \__regex_compile_raw:N #1
    \else:
      \if_meaning:w \__regex_compile_special:N #1
        \if_charcode:w ] #2
          \use_i:nn
        \fi:
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN \use_iii:nnn
      \fi:
    \fi:
    \use_i:nn
  }
\cs_new_protected:Npn \__regex_compile_range:Nw #1#2#3
  {
    \__regex_if_end_range:NNTF #2 #3
      {
        \if_int_compare:w `#1 > `#3 \exp_stop_f:
          \msg_error:nnee { regex } { range-backwards } {#1} {#3}
        \else:
          \tl_build_put_right:Ne \l__regex_build_tl
            {
              \if_int_compare:w `#1 = `#3 \exp_stop_f:
                \__regex_item_equal:n
              \else:
                \__regex_item_range:nn { \int_value:w `#1 }
              \fi:
              { \int_value:w `#3 }
            }
        \fi:
      }
      {
```

```
        \msg_warning:nnee { regex } { range-missing-end }
          {#1} { \c_backslash_str #3 }
        \tl_build_put_right:Ne \l__regex_build_tl
          {
            \__regex_item_equal:n { \int_value:w `#1 \exp_stop_f: }
            \__regex_item_equal:n { \int_value:w `- \exp_stop_f: }
          }
        #2#3
      }
  }
```

(*End of definition for* `\__regex_compile_range:Nw` *and* `\__regex_if_end_range:NNTF`.)

46.3.7 Character properties

`\__regex_compile_.:`
`\__regex_prop_.:`

In a class, the dot has no special meaning. Outside, insert `\__regex_prop_.:`, which matches any character or control sequence, and refuses -2 (end-marker).

```
\cs_new_protected:cpe { __regex_compile_.: }
  {
    \exp_not:N \__regex_if_in_class:TF
      { \__regex_compile_raw:N . }
      { \__regex_compile_one:n \exp_not:c { __regex_prop_.: } }
  }
\cs_new_protected:cpn { __regex_prop_.: }
  {
    \if_int_compare:w \l__regex_curr_char_int > - 2 \exp_stop_f:
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
```

(*End of definition for* `\__regex_compile_.:` *and* `\__regex_prop_.:`.)

`\__regex_compile_/d:`
`\__regex_compile_/D:`
`\__regex_compile_/h:`
`\__regex_compile_/H:`
`\__regex_compile_/s:`
`\__regex_compile_/S:`
`\__regex_compile_/v:`
`\__regex_compile_/V:`
`\__regex_compile_/w:`
`\__regex_compile_/W:`
`\__regex_compile_/N:`

The constants `\__regex_prop_d:`, *etc.* hold a list of tests which match the corresponding character class, and jump to the `\__regex_break_point:TF` marker. As for a normal character, we check for quantifiers.

```
\cs_set_protected:Npn \__regex_tmp:w #1#2
  {
    \cs_new_protected:cpe { __regex_compile_/#1: }
      { \__regex_compile_one:n \exp_not:c { __regex_prop_#1: } }
    \cs_new_protected:cpe { __regex_compile_/#2: }
      {
        \__regex_compile_one:n
          { \__regex_item_reverse:n { \exp_not:c { __regex_prop_#1: } } }
      }
  }
\__regex_tmp:w d D
\__regex_tmp:w h H
\__regex_tmp:w s S
\__regex_tmp:w v V
\__regex_tmp:w w W
\cs_new_protected:cpn { __regex_compile_/N: }
  { \__regex_compile_one:n \__regex_prop_N: }
```

(*End of definition for* `\__regex_compile_/d:` *and others.*)

46.3.8 Anchoring and simple assertions

__regex_compile_anchor_letter:NNN
__regex_compile_/A:
__regex_compile_/G:
__regex_compile_/Z:
__regex_compile_/z:
__regex_compile_/b:
__regex_compile_/B:
__regex_compile_^:
__regex_compile_$:

In modes where assertions are forbidden, anchors such as `\A` produce an error (`\A` is invalid in classes); otherwise they add an `\__regex_assertion:Nn` test as appropriate (the only negative assertion is `\B`). The test functions are defined later. The implementation for `$` and `^` is only different from `\A` etc because these are valid in a class.

```
\cs_new_protected:Npn \__regex_compile_anchor_letter:NNN #1#2#3
  {
    \__regex_if_in_class_or_catcode:TF { \__regex_compile_raw_error:N #1 }
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          { \__regex_assertion:Nn #2 {#3} }
      }
  }
\cs_new_protected:cpn { __regex_compile_/A: }
  { \__regex_compile_anchor_letter:NNN A \c_true_bool \__regex_A_test: }
\cs_new_protected:cpn { __regex_compile_/G: }
  { \__regex_compile_anchor_letter:NNN G \c_true_bool \__regex_G_test: }
\cs_new_protected:cpn { __regex_compile_/Z: }
  { \__regex_compile_anchor_letter:NNN Z \c_true_bool \__regex_Z_test: }
\cs_new_protected:cpn { __regex_compile_/z: }
  { \__regex_compile_anchor_letter:NNN z \c_true_bool \__regex_Z_test: }
\cs_new_protected:cpn { __regex_compile_/b: }
  { \__regex_compile_anchor_letter:NNN b \c_true_bool \__regex_b_test: }
\cs_new_protected:cpn { __regex_compile_/B: }
  { \__regex_compile_anchor_letter:NNN B \c_false_bool \__regex_b_test: }
\cs_set_protected:Npn \__regex_tmp:w #1#2
  {
    \cs_new_protected:cpn { __regex_compile_#1: }
      {
        \__regex_if_in_class_or_catcode:TF { \__regex_compile_raw:N #1 }
          {
            \tl_build_put_right:Nn \l__regex_build_tl
              { \__regex_assertion:Nn \c_true_bool {#2} }
          }
      }
  }
\exp_args:Ne \__regex_tmp:w { \iow_char:N \^ } { \__regex_A_test: }
\exp_args:Ne \__regex_tmp:w { \iow_char:N \$ } { \__regex_Z_test: }
```

(*End of definition for* `\__regex_compile_anchor_letter:NNN` *and others.*)

46.3.9 Character classes

__regex_compile_]:

Outside a class, right brackets have no meaning. In a class, change the mode ($m \to (m-15)/13$, truncated) to reflect the fact that we are leaving the class. Look for quantifiers, unless we are still in a class after leaving one (the case of `[...\cL[...]...]`). quantifiers.

```
\cs_new_protected:cpn { __regex_compile_]: }
  {
    \__regex_if_in_class:TF
      {
        \if_int_compare:w \l__regex_mode_int >
          \c__regex_catcode_in_class_mode_int
          \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
```

```
        \fi:
        \tex_advance:D \l__regex_mode_int - 15 \exp_stop_f:
        \tex_divide:D \l__regex_mode_int 13 \exp_stop_f:
        \if_int_odd:w \l__regex_mode_int \else:
          \exp_after:wN \__regex_compile_quantifier:w
        \fi:
      }
      { \__regex_compile_raw:N ] }
  }
```

(*End of definition for* `\__regex_compile_]:`.)

`\__regex_compile_[:` In a class, left brackets might introduce a POSIX character class, or mean nothing. Immediately following `\c`⟨*category*⟩, we must insert the appropriate catcode test, then parse the class; we pre-expand the catcode as an optimization. Otherwise (modes 0, −2 and −6) just parse the class. The mode is updated later.

```
\cs_new_protected:cpn { __regex_compile_[: }
  {
    \__regex_if_in_class:TF
      { \__regex_compile_class_posix_test:w }
      {
        \__regex_if_within_catcode:TF
          {
            \exp_after:wN \__regex_compile_class_catcode:w
              \int_use:N \l__regex_catcodes_int ;
          }
          { \__regex_compile_class_normal:w }
      }
  }
```

(*End of definition for* `\__regex_compile_[:`.)

`\__regex_compile_class_normal:w` In the "normal" case, we insert `\__regex_class:NnnnN` ⟨*boolean*⟩ in the compiled code. The ⟨*boolean*⟩ is true for positive classes, and false for negative classes, characterized by a leading `^`. The auxiliary `\__regex_compile_class:TFNN` also checks for a leading `]` which has a special meaning.

```
\cs_new_protected:Npn \__regex_compile_class_normal:w
  {
    \__regex_compile_class:TFNN
      { \__regex_class:NnnnN \c_true_bool }
      { \__regex_class:NnnnN \c_false_bool }
  }
```

(*End of definition for* `\__regex_compile_class_normal:w`.)

`\__regex_compile_class_catcode:w` This function is called for a left bracket in modes 2 or 6 (catcode test, and catcode test within a class). In mode 2 the whole construction needs to be put in a class (like single character). Then determine if the class is positive or negative, inserting `\__regex_item_catcode:nT` or the `reverse` variant as appropriate, each with the current catcodes bitmap `#1` as an argument, and reset the catcodes.

```
\cs_new_protected:Npn \__regex_compile_class_catcode:w #1;
  {
    \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
      \tl_build_put_right:Nn \l__regex_build_tl
```

```
        { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
    \fi:
    \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
    \__regex_compile_class:TFNN
      { \__regex_item_catcode:nT {#1} }
      { \__regex_item_catcode_reverse:nT {#1} }
  }
```

(*End of definition for* `\__regex_compile_class_catcode:w`.)

`\__regex_compile_class:TFNN`
`\__regex_compile_class:NN`

If the first character is `^`, then the class is negative (use `#2`), otherwise it is positive (use `#1`). If the next character is a right bracket, then it should be changed to a raw one.

```
\cs_new_protected:Npn \__regex_compile_class:TFNN #1#2#3#4
  {
    \l__regex_mode_int = \int_value:w \l__regex_mode_int 3 \exp_stop_f:
    \__regex_two_if_eq:NNNNTF #3 #4 \__regex_compile_special:N ^
      {
        \tl_build_put_right:Nn \l__regex_build_tl { #2 { \if_false: } \fi: }
        \__regex_compile_class:NN
      }
      {
        \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
        \__regex_compile_class:NN #3 #4
      }
  }
\cs_new_protected:Npn \__regex_compile_class:NN #1#2
  {
    \token_if_eq_charcode:NNTF #2 ]
      { \__regex_compile_raw:N #2 }
      { #1 #2 }
  }
```

(*End of definition for* `\__regex_compile_class:TFNN` *and* `\__regex_compile_class:NN`.)

`\__regex_compile_class_posix_test:w`
`\__regex_compile_class_posix:NNNNw`
`\__regex_compile_class_posix_loop:w`
`\__regex_compile_class_posix_end:w`

Here we check for a syntax such as `[:alpha:]`. We also detect `[=` and `[.` which have a meaning in POSIX regular expressions, but are not implemented in l3regex. In case we see `[:`, grab raw characters until hopefully reaching `:]`. If that's missing, or the POSIX class is unknown, abort. If all is right, add the test to the current class, with an extra `\__regex_item_reverse:n` for negative classes (we make sure to wrap its argument in braces otherwise `\regex_show:N` would not recognize the regex as valid).

```
\cs_new_protected:Npn \__regex_compile_class_posix_test:w #1#2
  {
    \token_if_eq_meaning:NNT \__regex_compile_special:N #1
      {
        \str_case:nn { #2 }
          {
            : { \__regex_compile_class_posix:NNNNw }
            = {
                \msg_warning:nne { regex }
                  { posix-unsupported } { = }
              }
            . {
                \msg_warning:nne { regex }
                  { posix-unsupported } { . }
```

```
            }
        }
      }
    \__regex_compile_raw:N [ #1 #2
  }
\cs_new_protected:Npn \__regex_compile_class_posix:NNNNw #1#2#3#4#5#6
  {
    \__regex_two_if_eq:NNNNTF #5 #6 \__regex_compile_special:N ^
      {
        \bool_set_false:N \l__regex_internal_bool
        \__kernel_tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
          \__regex_compile_class_posix_loop:w
      }
      {
        \bool_set_true:N \l__regex_internal_bool
        \__kernel_tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
          \__regex_compile_class_posix_loop:w #5 #6
      }
  }
\cs_new:Npn \__regex_compile_class_posix_loop:w #1#2
  {
    \token_if_eq_meaning:NNTF \__regex_compile_raw:N #1
      { #2 \__regex_compile_class_posix_loop:w }
      { \if_false: { \fi: } \__regex_compile_class_posix_end:w #1 #2 }
  }
\cs_new_protected:Npn \__regex_compile_class_posix_end:w #1#2#3#4
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N :
      { \__regex_two_if_eq:NNNNTF #3 #4 \__regex_compile_special:N ] }
      { \use_ii:nn }
      {
        \cs_if_exist:cTF { __regex_posix_ \l__regex_internal_a_tl : }
          {
            \__regex_compile_one:n
              {
                \bool_if:NTF \l__regex_internal_bool \use:n \__regex_item_reverse:n
                { \exp_not:c { __regex_posix_ \l__regex_internal_a_tl : } }
              }
          }
          {
            \msg_warning:nne { regex } { posix-unknown }
              { \l__regex_internal_a_tl }
            \__regex_compile_abort_tokens:e
              {
                [: \bool_if:NF \l__regex_internal_bool { ^ }
                \l__regex_internal_a_tl :]
              }
          }
      }
      {
        \msg_error:nnee { regex } { posix-missing-close }
          { [: \l__regex_internal_a_tl } { #2 #4 }
        \__regex_compile_abort_tokens:e { [: \l__regex_internal_a_tl }
        #1 #2 #3 #4
```

```
      }
  }
```

(*End of definition for* `\__regex_compile_class_posix_test:w` *and others.*)

46.3.10 Groups and alternations

`\__regex_compile_group_begin:N`
`\__regex_compile_group_end:`

The contents of a regex group are turned into compiled code in `\l__regex_build_tl`, which ends up with items of the form `\__regex_branch:n {`⟨*concatenation*⟩`}`. This construction is done using `\tl_build_...` functions within a TeX group, which automatically makes sure that options (case-sensitivity and default catcode) are reset at the end of the group. The argument `#1` is `\__regex_group:nnnN` or a variant thereof. A small subtlety to support `\cL(abc)` as a shorthand for `(\cLa\cLb\cLc)`: exit any pending catcode test, save the category code at the start of the group as the default catcode for that group, and make sure that the catcode is restored to the default outside the group.

```
\cs_new_protected:Npn \__regex_compile_group_begin:N #1
  {
    \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
    \__regex_mode_quit_c:
    \group_begin:
      \tl_build_begin:N \l__regex_build_tl
      \int_set_eq:NN \l__regex_default_catcodes_int \l__regex_catcodes_int
      \int_incr:N \l__regex_group_level_int
      \tl_build_put_right:Nn \l__regex_build_tl
        { \__regex_branch:n { \if_false: } \fi: }
  }
\cs_new_protected:Npn \__regex_compile_group_end:
  {
    \if_int_compare:w \l__regex_group_level_int > \c_zero_int
        \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
        \tl_build_end:N \l__regex_build_tl
        \exp_args:NNNe
      \group_end:
      \tl_build_put_right:Nn \l__regex_build_tl { \l__regex_build_tl }
      \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
      \exp_after:wN \__regex_compile_quantifier:w
    \else:
      \msg_warning:nn { regex } { extra-rparen }
      \exp_after:wN \__regex_compile_raw:N \exp_after:wN )
    \fi:
  }
```

(*End of definition for* `\__regex_compile_group_begin:N` *and* `\__regex_compile_group_end:`*.*)

`\__regex_compile_(:`

In a class, parentheses are not special. In a catcode test inside a class, a left parenthesis gives an error, to catch `[a\cL(bcd)e]`. Otherwise check for a `?`, denoting special groups, and run the code for the corresponding special group.

```
\cs_new_protected:cpn { __regex_compile_(: }
  {
    \__regex_if_in_class:TF { \__regex_compile_raw:N ( }
      {
        \if_int_compare:w \l__regex_mode_int =
```

```
        \c__regex_catcode_in_class_mode_int
        \msg_error:nn { regex } { c-lparen-in-class }
        \exp_after:wN \__regex_compile_raw:N \exp_after:wN (
      \else:
        \exp_after:wN \__regex_compile_lparen:w
      \fi:
    }
  }
\cs_new_protected:Npn \__regex_compile_lparen:w #1#2#3#4
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N ?
      {
        \cs_if_exist_use:cF
          { __regex_compile_special_group_\token_to_str:N #4 :w }
          {
            \msg_warning:nne { regex } { special-group-unknown }
              { (? #4 }
            \__regex_compile_group_begin:N \__regex_group:nnnN
              \__regex_compile_raw:N ? #3 #4
          }
      }
      {
        \__regex_compile_group_begin:N \__regex_group:nnnN
          #1 #2 #3 #4
      }
  }
```

(*End of definition for* `\__regex_compile_(:`.)

`\__regex_compile_|:` In a class, the pipe is not special. Otherwise, end the current branch and open another one.

```
\cs_new_protected:cpn { __regex_compile_|: }
  {
    \__regex_if_in_class:TF { \__regex_compile_raw:N | }
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          { \if_false: { \fi: } \__regex_branch:n { \if_false: } \fi: }
      }
  }
```

(*End of definition for* `\__regex_compile_|:`.)

`\__regex_compile_):` Within a class, parentheses are not special. Outside, close a group.

```
\cs_new_protected:cpn { __regex_compile_): }
  {
    \__regex_if_in_class:TF { \__regex_compile_raw:N ) }
      { \__regex_compile_group_end: }
  }
```

(*End of definition for* `\__regex_compile_):`.)

`\__regex_compile_special_group_::w`
`\__regex_compile_special_group_|:w`

Non-capturing, and resetting groups are easy to take care of during compilation; for those groups, the harder parts come when building.

```
\cs_new_protected:cpn { __regex_compile_special_group_::w }
  { \__regex_compile_group_begin:N \__regex_group_no_capture:nnnN }
```

```
\cs_new_protected:cpn { __regex_compile_special_group_|:w }
  { \__regex_compile_group_begin:N \__regex_group_resetting:nnnN }
```

(*End of definition for* `\__regex_compile_special_group_::w` *and* `\__regex_compile_special_group_-|:w`*.*)

`\__regex_compile_special_group_i:w`
`\__regex_compile_special_group_-:w`

The match can be made case-insensitive by setting the option with `(?i)`; the original behaviour is restored by `(?-i)`. This is the only supported option.

```
\cs_new_protected:Npn \__regex_compile_special_group_i:w #1#2
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N )
      {
        \cs_set:Npn \__regex_item_equal:n
          { \__regex_item_caseless_equal:n }
        \cs_set:Npn \__regex_item_range:nn
          { \__regex_item_caseless_range:nn }
      }
      {
        \msg_warning:nne { regex } { unknown-option } { (?i #2 }
        \__regex_compile_raw:N (
        \__regex_compile_raw:N ?
        \__regex_compile_raw:N i
        #1 #2
      }
  }
\cs_new_protected:cpn { __regex_compile_special_group_-:w } #1#2#3#4
  {
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_raw:N i
      { \__regex_two_if_eq:NNNNTF #3 #4 \__regex_compile_special:N ) }
      { \use_ii:nn }
      {
        \cs_set:Npn \__regex_item_equal:n
          { \__regex_item_caseful_equal:n }
        \cs_set:Npn \__regex_item_range:nn
          { \__regex_item_caseful_range:nn }
      }
      {
        \msg_warning:nne { regex } { unknown-option } { (?-#2#4 }
        \__regex_compile_raw:N (
        \__regex_compile_raw:N ?
        \__regex_compile_raw:N -
        #1 #2 #3 #4
      }
  }
```

(*End of definition for* `\__regex_compile_special_group_i:w` *and* `\__regex_compile_special_group_--:w`*.*)

46.3.11 Catcodes and csnames

`\__regex_compile_/c:`
`\__regex_compile_c_test:NN`

The `\c` escape sequence can be followed by a capital letter representing a character category, by a left bracket which starts a list of categories, or by a brace group holding a regular expression for a control sequence name. Otherwise, raise an error.

```
\cs_new_protected:cpn { __regex_compile_/c: }
```

```
    { \__regex_chk_c_allowed:T { \__regex_compile_c_test:NN } }
\cs_new_protected:Npn \__regex_compile_c_test:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
      {
        \int_if_exist:cTF { c__regex_catcode_#2_int }
          {
            \int_set_eq:Nc \l__regex_catcodes_int
              { c__regex_catcode_#2_int }
            \l__regex_mode_int
              = \if_case:w \l__regex_mode_int
                  \c__regex_catcode_mode_int
                \else:
                  \c__regex_catcode_in_class_mode_int
                \fi:
            \token_if_eq_charcode:NNT C #2 { \__regex_compile_c_C:NN }
          }
      }
      { \cs_if_exist_use:cF { __regex_compile_c_#2:w } }
          {
            \msg_error:nne { regex } { c-missing-category } {#2}
            #1 #2
          }
  }
```

(*End of definition for* `\__regex_compile_/c:` *and* `\__regex_compile_c_test:NN`.)

`\__regex_compile_c_C:NN`

If `\cC` is not followed by `.` or `(...)` then complain because that construction cannot match anything, except in cases like `\cC[\c{...}]`, where it has no effect.

```
\cs_new_protected:Npn \__regex_compile_c_C:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
      {
        \token_if_eq_charcode:NNTF #2 .
          { \use_none:n }
          { \token_if_eq_charcode:NNF #2 ( } % )
      }
      { \use:n }
    { \msg_error:nnn { regex } { c-C-invalid } {#2} }
    #1 #2
  }
```

(*End of definition for* `\__regex_compile_c_C:NN`.)

`\__regex_compile_c_[:w`
`\__regex_compile_c_lbrack_loop:NN`
`\__regex_compile_c_lbrack_add:N`
`\__regex_compile_c_lbrack_end:`

When encountering `\c[`, the task is to collect uppercase letters representing character categories. First check for `^` which negates the list of category codes.

```
\cs_new_protected:cpn { __regex_compile_c_[:w } #1#2
  {
    \l__regex_mode_int
      = \if_case:w \l__regex_mode_int
          \c__regex_catcode_mode_int
        \else:
          \c__regex_catcode_in_class_mode_int
        \fi:
```

```
    \int_zero:N \l__regex_catcodes_int
    \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N ^
      {
        \bool_set_false:N \l__regex_catcodes_bool
        \__regex_compile_c_lbrack_loop:NN
      }
      {
        \bool_set_true:N \l__regex_catcodes_bool
        \__regex_compile_c_lbrack_loop:NN
        #1 #2
      }
  }
\cs_new_protected:Npn \__regex_compile_c_lbrack_loop:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
      {
        \int_if_exist:cTF { c__regex_catcode_#2_int }
          {
            \exp_args:Nc \__regex_compile_c_lbrack_add:N
              { c__regex_catcode_#2_int }
            \__regex_compile_c_lbrack_loop:NN
          }
      }
      {
        \token_if_eq_charcode:NNTF #2 ]
          { \__regex_compile_c_lbrack_end: }
      }
          {
            \msg_error:nne { regex } { c-missing-rbrack } {#2}
            \__regex_compile_c_lbrack_end:
            #1 #2
          }
  }
\cs_new_protected:Npn \__regex_compile_c_lbrack_add:N #1
  {
    \if_int_odd:w \__regex_int_eval:w \l__regex_catcodes_int / #1 \scan_stop:
    \else:
      \int_add:Nn \l__regex_catcodes_int {#1}
    \fi:
  }
\cs_new_protected:Npn \__regex_compile_c_lbrack_end:
  {
    \if_meaning:w \c_false_bool \l__regex_catcodes_bool
      \int_set:Nn \l__regex_catcodes_int
        { \c__regex_all_catcodes_int - \l__regex_catcodes_int }
    \fi:
  }
```

(*End of definition for* `\__regex_compile_c_[:w` *and others.*)

`\__regex_compile_c_{:` The case of a left brace is easy, based on what we have done so far: in a group, compile the regular expression, after changing the mode to forbid nesting `\c`. Additionally, disable submatch tracking since groups don't escape the scope of `\c{...}`.

```
\cs_new_protected:cpn { __regex_compile_c_ \c_left_brace_str :w }
```

```
  {
    \__regex_compile:w
      \__regex_disable_submatches:
      \l__regex_mode_int
        = \if_case:w \l__regex_mode_int
            \c__regex_cs_mode_int
          \else:
            \c__regex_cs_in_class_mode_int
          \fi:
  }
```

(*End of definition for* `\__regex_compile_c_{:`.)

`\__regex_compile_{:`

We forbid unescaped left braces inside a `\c{...}` escape because they otherwise lead to the confusing question of whether the first right brace in `\c{{}x}` should end `\c` or whether one should match braces.

```
\cs_new_protected:cpn { __regex_compile_ \c_left_brace_str : }
  {
    \__regex_if_in_cs:TF
      { \msg_error:nnn { regex } { cu-lbrace } { c } }
      { \exp_after:wN \__regex_compile_raw:N \c_left_brace_str }
  }
```

(*End of definition for* `\__regex_compile_{:`.)

`\l__regex_cs_flag`
`\__regex_compile_}:`
`\__regex_compile_end_cs:`
`\__regex_compile_cs_aux:Nn`
`\__regex_compile_cs_aux:NNnnnN`

Non-escaped right braces are only special if they appear when compiling the regular expression for a csname, but not within a class: `\c{[{}]}` matches the control sequences `\{` and `\}`. So, end compiling the inner regex (this closes any dangling class or group). Then insert the corresponding test in the outer regex. As an optimization, if the control sequence test simply consists of several explicit possibilities (branches) then use `\__regex_item_exact_cs:n` with an argument consisting of all possibilities separated by `\scan_stop:`.

```
\flag_new:N \l__regex_cs_flag
\cs_new_protected:cpn { __regex_compile_ \c_right_brace_str : }
  {
    \__regex_if_in_cs:TF
      { \__regex_compile_end_cs: }
      { \exp_after:wN \__regex_compile_raw:N \c_right_brace_str }
  }
\cs_new_protected:Npn \__regex_compile_end_cs:
  {
    \__regex_compile_end:
    \flag_clear:N \l__regex_cs_flag
    \__kernel_tl_set:Nx \l__regex_internal_a_tl
      {
        \exp_after:wN \__regex_compile_cs_aux:Nn \l__regex_internal_regex
        \q__regex_nil \q__regex_nil \q__regex_recursion_stop
      }
    \exp_args:Ne \__regex_compile_one:n
      {
        \flag_if_raised:NTF \l__regex_cs_flag
          { \__regex_item_cs:n { \exp_not:o \l__regex_internal_regex } }
          {
            \__regex_item_exact_cs:n
```

```
            { \tl_tail:N \l__regex_internal_a_tl }
          }
      }
  }
\cs_new:Npn \__regex_compile_cs_aux:Nn #1#2
  {
    \cs_if_eq:NNTF #1 \__regex_branch:n
      {
        \scan_stop:
        \__regex_compile_cs_aux:NNnnnN #2
        \q__regex_nil \q__regex_nil \q__regex_nil
        \q__regex_nil \q__regex_nil \q__regex_nil \q__regex_recursion_stop
        \__regex_compile_cs_aux:Nn
      }
      {
        \__regex_quark_if_nil:NF #1 { \flag_ensure_raised:N \l__regex_cs_flag }
        \__regex_use_none_delimit_by_q_recursion_stop:w
      }
  }
\cs_new:Npn \__regex_compile_cs_aux:NNnnnN #1#2#3#4#5#6
  {
    \bool_lazy_all:nTF
      {
        { \cs_if_eq_p:NN #1 \__regex_class:NnnnN }
        {#2}
        { \tl_if_head_eq_meaning_p:nN {#3} \__regex_item_caseful_equal:n }
        { \int_compare_p:nNn { \tl_count:n {#3} } = { 2 } }
        { \int_compare_p:nNn {#5} = \c_zero_int }
      }
      {
        \prg_replicate:nn {#4}
          { \char_generate:nn { \use_ii:nn #3 } {12} }
        \__regex_compile_cs_aux:NNnnnN
      }
      {
        \__regex_quark_if_nil:NF #1
          {
            \flag_ensure_raised:N \l__regex_cs_flag
            \__regex_use_i_delimit_by_q_recursion_stop:nw
          }
        \__regex_use_none_delimit_by_q_recursion_stop:w
      }
  }
```

(*End of definition for* `\l__regex_cs_flag` *and others.*)

46.3.12 Raw token lists with \u

`\__regex_compile_/u:`

The `\u` escape is invalid in classes and directly following a catcode test. Otherwise test for a following `r` (for `\ur`), and call an auxiliary responsible for finding the variable name.

```
\cs_new_protected:cpn { __regex_compile_/u: } #1#2
  {
    \__regex_if_in_class_or_catcode:TF
      { \__regex_compile_raw_error:N u #1 #2 }
```

```
      {
        \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_raw:N r
          { \__regex_compile_u_brace:NNN \__regex_compile_ur_end: }
          { \__regex_compile_u_brace:NNN \__regex_compile_u_end: #1 #2 }
      }
  }
```

(*End of definition for* `\__regex_compile_/u:`.)

`\__regex_compile_u_brace:NNN`

This enforces the presence of a left brace, then starts a loop to find the variable name.

```
\cs_new:Npn \__regex_compile_u_brace:NNN #1#2#3
  {
    \__regex_two_if_eq:NNNNTF #2 #3 \__regex_compile_special:N \c_left_brace_str
      {
        \tl_set:Nn \l__regex_internal_b_tl {#1}
        \__kernel_tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
        \__regex_compile_u_loop:NN
      }
      {
        \msg_error:nn { regex } { u-missing-lbrace }
        \token_if_eq_meaning:NNTF #1 \__regex_compile_ur_end:
          { \__regex_compile_raw:N u \__regex_compile_raw:N r }
          { \__regex_compile_raw:N u }
        #2 #3
      }
  }
```

(*End of definition for* `\__regex_compile_u_brace:NNN`.)

`\__regex_compile_u_loop:NN`

We collect the characters for the argument of `\u` within an `e`-expanding assignment. In principle we could just wait to encounter a right brace, but this is unsafe: if the right brace was missing, then we would reach the end-markers of the regex, and continue, leading to obscure fatal errors. Instead, we only allow raw and special characters, and stop when encountering a special right brace, any escaped character, or the end-marker.

```
\cs_new:Npn \__regex_compile_u_loop:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
      { #2 \__regex_compile_u_loop:NN }
      {
        \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
          {
            \exp_after:wN \token_if_eq_charcode:NNTF \c_right_brace_str #2
              { \if_false: { \fi: } \l__regex_internal_b_tl }
              {
                \if_charcode:w \c_left_brace_str #2
                  \msg_expandable_error:nnn { regex } { cu-lbrace } { u }
                \else:
                  #2
                \fi:
                \__regex_compile_u_loop:NN
              }
          }
          {
            \if_false: { \fi: }
```

```
            \msg_error:nne { regex } { u-missing-rbrace } {#2}
            \l__regex_internal_b_tl
            #1 #2
          }
      }
  }
```

(*End of definition for* `\__regex_compile_u_loop:NN`.)

`\__regex_compile_ur_end:` `\__regex_compile_ur:n` `\__regex_compile_ur_aux:w`

For the `\ur{...}` construction, once we have extracted the variable's name, we replace all groups by non-capturing groups in the compiled regex (passed as the argument of `\__regex_compile_ur:n`). If that has a single branch (namely `\tl_if_empty:oTF` is false) and there is no quantifier, then simply insert the contents of this branch (obtained by `\use_ii:nn`, which is expanded later). In all other cases, insert a non-capturing group and look for quantifiers to determine the number of repetition etc.

```
\cs_new_protected:Npn \__regex_compile_ur_end:
  {
    \group_begin:
      \cs_set:Npn \__regex_group:nnnN { \__regex_group_no_capture:nnnN }
      \cs_set:Npn \__regex_group_resetting:nnnN { \__regex_group_no_capture:nnnN }
      \exp_args:NNe
    \group_end:
    \__regex_compile_ur:n { \use:c { \l__regex_internal_a_tl } }
  }
\cs_new_protected:Npn \__regex_compile_ur:n #1
  {
    \tl_if_empty:oTF { \__regex_compile_ur_aux:w #1 {} ? ? \q__regex_nil }
      { \__regex_compile_if_quantifier:TFw }
      { \use_i:nn }
          {
            \tl_build_put_right:Nn \l__regex_build_tl
              { \__regex_group_no_capture:nnnN { \if_false: } \fi: #1 }
            \__regex_compile_quantifier:w
          }
          { \tl_build_put_right:Nn \l__regex_build_tl { \use_ii:nn #1 } }
  }
\cs_new:Npn \__regex_compile_ur_aux:w \__regex_branch:n #1#2#3 \q__regex_nil {#2}
```

(*End of definition for* `\__regex_compile_ur_end:`, `\__regex_compile_ur:n`, *and* `\__regex_compile_ur_aux:w`.)

`\__regex_compile_u_end:` `\__regex_compile_u_payload:`

Once we have extracted the variable's name, we check for quantifiers, in which case we set up a non-capturing group with a single branch. Inside this branch (we omit it and the group if there is no quantifier), `\__regex_compile_u_payload:` puts the right tests corresponding to the contents of the variable, which we store in `\l__regex_internal_a_tl`. The behaviour of `\u` then depends on whether we are within a `\c{...}` escape (in this case, the variable is turned to a string), or not.

```
\cs_new_protected:Npn \__regex_compile_u_end:
  {
    \__regex_compile_if_quantifier:TFw
      {
        \tl_build_put_right:Nn \l__regex_build_tl
          {
            \__regex_group_no_capture:nnnN { \if_false: } \fi:
```

```
            \__regex_branch:n { \if_false: } \fi:
          }
        \__regex_compile_u_payload:
        \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
        \__regex_compile_quantifier:w
      }
      { \__regex_compile_u_payload: }
  }
\cs_new_protected:Npn \__regex_compile_u_payload:
  {
    \tl_set:Nv \l__regex_internal_a_tl { \l__regex_internal_a_tl }
    \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
      \__regex_compile_u_not_cs:
    \else:
      \__regex_compile_u_in_cs:
    \fi:
  }
```

(*End of definition for `\__regex_compile_u_end:` and `\__regex_compile_u_payload:`.*)

`\__regex_compile_u_in_cs:` When `\u` appears within a control sequence, we convert the variable to a string with escaped spaces. Then for each character insert a class matching exactly that character, once.

```
\cs_new_protected:Npn \__regex_compile_u_in_cs:
  {
    \__kernel_tl_gset:Nx \g__regex_internal_tl
      {
        \exp_args:No \__kernel_str_to_other_fast:n
          { \l__regex_internal_a_tl }
      }
    \tl_build_put_right:Ne \l__regex_build_tl
      {
        \tl_map_function:NN \g__regex_internal_tl
          \__regex_compile_u_in_cs_aux:n
      }
  }
\cs_new:Npn \__regex_compile_u_in_cs_aux:n #1
  {
    \__regex_class:NnnnN \c_true_bool
      { \__regex_item_caseful_equal:n { \int_value:w `#1 } }
      { 1 } { 0 } \c_false_bool
  }
```

(*End of definition for `\__regex_compile_u_in_cs:`.*)

`\__regex_compile_u_not_cs:` In mode 0, the `\u` escape adds one state to the NFA for each token in `\l__regex_internal_a_tl`. If a given ⟨*token*⟩ is a control sequence, then insert a string comparison test, otherwise, `\__regex_item_exact:nn` which compares catcode and character code.

```
\cs_new_protected:Npn \__regex_compile_u_not_cs:
  {
    \tl_analysis_map_inline:Nn \l__regex_internal_a_tl
      {
        \tl_build_put_right:Ne \l__regex_build_tl
          {
```

```
            \__regex_class:NnnnN \c_true_bool
              {
                \if_int_compare:w "##3 = \c_zero_int
                  \__regex_item_exact_cs:n
                    { \exp_after:wN \cs_to_str:N ##1 }
                \else:
                  \__regex_item_exact:nn { \int_value:w "##3 } { ##2 }
                \fi:
              }
              { 1 } { 0 } \c_false_bool
          }
      }
  }
```

(*End of definition for* `\__regex_compile_u_not_cs:`.)

46.3.13 Other

`\__regex_compile_/K:`

The `\K` control sequence is currently the only "command", which performs some action, rather than matching something. It is allowed in the same contexts as `\b`. At the compilation stage, we leave it as a single control sequence, defined later.

```
\cs_new_protected:cpn { __regex_compile_/K: }
  {
    \int_compare:nNnTF \l__regex_mode_int = \c__regex_outer_mode_int
      { \tl_build_put_right:Nn \l__regex_build_tl { \__regex_command_K: } }
      { \__regex_compile_raw_error:N K }
  }
```

(*End of definition for* `\__regex_compile_/K:`.)

46.3.14 Showing regexes

`\__regex_clean_bool:n`
`\__regex_clean_int:n`
`\__regex_clean_int_aux:N`
`\__regex_clean_regex:n`
`\__regex_clean_regex_loop:w`
`\__regex_clean_branch:n`
`\__regex_clean_branch_loop:n`
`\__regex_clean_assertion:Nn`
`\__regex_clean_class:NnnnN`
`\__regex_clean_group:nnnN`
`\__regex_clean_class:n`
`\__regex_clean_class_loop:nnn`
`\__regex_clean_exact_cs:n`
`\__regex_clean_exact_cs:w`

Before showing a regex we check that it is "clean" in the sense that it has the correct internal structure. We do this (in the implementation of `\regex_show:N` and `\regex_log:N`) by comparing it with a cleaned-up version of the same regex. Along the way we also need similar functions for other types: all `\__regex_clean_`⟨*type*⟩`:n` functions produce valid ⟨*type*⟩ tokens (bool, explicit integer, etc.) from arbitrary input, and the output coincides with the input if that was valid.

```
\cs_new:Npn \__regex_clean_bool:n #1
  {
    \tl_if_single:nTF {#1}
      { \bool_if:NTF #1 \c_true_bool \c_false_bool }
      { \c_true_bool }
  }
\cs_new:Npn \__regex_clean_int:n #1
  {
    \tl_if_head_eq_meaning:nNTF {#1} -
      { - \exp_args:No \__regex_clean_int:n { \use_none:n #1 } }
      { \int_eval:n { 0 \str_map_function:nN {#1} \__regex_clean_int_aux:N } }
  }
\cs_new:Npn \__regex_clean_int_aux:N #1
  {
    \if_int_compare:w \c_one_int < 1 #1 ~
```

```
      #1
    \else:
      \str_map_break:n
    \fi:
  }
\cs_new:Npn \__regex_clean_regex:n #1
  {
    \__regex_clean_regex_loop:w #1
    \__regex_branch:n { \q_recursion_tail } \q_recursion_stop
  }
\cs_new:Npn \__regex_clean_regex_loop:w #1 \__regex_branch:n #2
  {
    \quark_if_recursion_tail_stop:n {#2}
    \__regex_branch:n { \__regex_clean_branch:n {#2} }
    \__regex_clean_regex_loop:w
  }
\cs_new:Npn \__regex_clean_branch:n #1
  {
    \__regex_clean_branch_loop:n #1
    ? ? ? ? ? ? \prg_break_point:
  }
\cs_new:Npn \__regex_clean_branch_loop:n #1
  {
    \tl_if_single:nF {#1} \prg_break:
    \token_case_meaning:NnF #1
      {
        \__regex_command_K: { #1 \__regex_clean_branch_loop:n }
        \__regex_assertion:Nn { #1 \__regex_clean_assertion:Nn }
        \__regex_class:NnnnN { #1 \__regex_clean_class:NnnnN }
        \__regex_group:nnnN { #1 \__regex_clean_group:nnnN }
        \__regex_group_no_capture:nnnN { #1 \__regex_clean_group:nnnN }
        \__regex_group_resetting:nnnN { #1 \__regex_clean_group:nnnN }
      }
      \prg_break:
  }
\cs_new:Npn \__regex_clean_assertion:Nn #1#2
  {
    \__regex_clean_bool:n {#1}
    \tl_if_single:nF {#2} { { \__regex_A_test: } \prg_break: }
    \token_case_meaning:NnTF #2
      {
        \__regex_A_test: { }
        \__regex_G_test: { }
        \__regex_Z_test: { }
        \__regex_b_test: { }
      }
      { {#2} }
      { { \__regex_A_test: } \prg_break: }
    \__regex_clean_branch_loop:n
  }
\cs_new:Npn \__regex_clean_class:NnnnN #1#2#3#4#5
  {
    \__regex_clean_bool:n {#1}
    { \__regex_clean_class:n {#2} }
```

```
    { \int_max:nn \c_zero_int { \__regex_clean_int:n {#3} } }
    { \int_max:nn { -\c_one_int } { \__regex_clean_int:n {#4} } }
    \__regex_clean_bool:n {#5}
    \__regex_clean_branch_loop:n
  }
\cs_new:Npn \__regex_clean_group:nnnN #1#2#3#4
  {
    { \__regex_clean_regex:n {#1} }
    { \int_max:nn \c_zero_int { \__regex_clean_int:n {#2} } }
    { \int_max:nn { -\c_one_int } { \__regex_clean_int:n {#3} } }
    \__regex_clean_bool:n {#4}
    \__regex_clean_branch_loop:n
  }
\cs_new:Npn \__regex_clean_class:n #1
  { \__regex_clean_class_loop:nnn #1 ????? \prg_break_point: }
```

When cleaning a class there are many cases, including a dozen or so like `\__regex_prop_d:` or `\__regex_posix_alpha:`. To avoid listing all of them we allow any command that starts with the 13 characters `__regex_prop_` or `__regex_posix` (handily these have the same length, except for the trailing underscore).

```
\cs_new:Npn \__regex_clean_class_loop:nnn #1#2#3
  {
    \tl_if_single:nF {#1} \prg_break:
    \token_case_meaning:NnTF #1
      {
        \__regex_item_cs:n { #1 { \__regex_clean_regex:n {#2} } }
        \__regex_item_exact_cs:n { #1 { \__regex_clean_exact_cs:n {#2} } }
        \__regex_item_caseful_equal:n { #1 { \__regex_clean_int:n {#2} } }
        \__regex_item_caseless_equal:n { #1 { \__regex_clean_int:n {#2} } }
        \__regex_item_reverse:n { #1 { \__regex_clean_class:n {#2} } }
      }
      { \__regex_clean_class_loop:nnn {#3} }
      {
        \token_case_meaning:NnTF #1
          {
            \__regex_item_caseful_range:nn { }
            \__regex_item_caseless_range:nn { }
            \__regex_item_exact:nn { }
          }
          {
            #1 { \__regex_clean_int:n {#2} } { \__regex_clean_int:n {#3} }
            \__regex_clean_class_loop:nnn
          }
          {
            \token_case_meaning:NnTF #1
              {
                \__regex_item_catcode:nT { }
                \__regex_item_catcode_reverse:nT { }
              }
              {
                #1 { \__regex_clean_int:n {#2} } { \__regex_clean_class:n {#3} }
                \__regex_clean_class_loop:nnn
              }
              {
```

```
              \exp_args:Ne \str_case:nnTF
                {
                  \exp_args:Ne \str_range:nnn
                    { \cs_to_str:N #1 } \c_one_int { 13 }
                }
                {
                  { __regex_prop_ } { }
                  { __regex_posix } { }
                }
                {
                  #1
                  \__regex_clean_class_loop:nnn {#2} {#3}
                }
                \prg_break:
            }
        }
    }
  }
\cs_new:Npn \__regex_clean_exact_cs:n #1
  {
    \exp_last_unbraced:Nf \use_none:n
      {
        \__regex_clean_exact_cs:w #1
        \scan_stop: \q_recursion_tail \scan_stop:
        \q_recursion_stop
      }
  }
\cs_new:Npn \__regex_clean_exact_cs:w #1 \scan_stop:
  {
    \quark_if_recursion_tail_stop:n {#1}
    \scan_stop: \tl_to_str:n {#1}
    \__regex_clean_exact_cs:w
  }
```

(*End of definition for* `\__regex_clean_bool:n` *and others.*)

`\__regex_show:N` Within a group and within `\tl_build_begin:N` … `\tl_build_end:N` we redefine all the function that can appear in a compiled regex, then run the regex. The result stored in `\l__regex_internal_a_tl` is then meant to be shown.

```
\cs_new_protected:Npn \__regex_show:N #1
  {
    \group_begin:
      \tl_build_begin:N \l__regex_build_tl
      \cs_set_protected:Npn \__regex_branch:n
        {
          \seq_pop_right:NN \l__regex_show_prefix_seq
            \l__regex_internal_a_tl
          \__regex_show_one:n { +-branch }
          \seq_put_right:No \l__regex_show_prefix_seq
            \l__regex_internal_a_tl
          \use:n
        }
      \cs_set_protected:Npn \__regex_group:nnnN
        { \__regex_show_group_aux:nnnnN { } }
```

```
      \cs_set_protected:Npn \__regex_group_no_capture:nnnN
        { \__regex_show_group_aux:nnnnN { ~(no~capture) } }
      \cs_set_protected:Npn \__regex_group_resetting:nnnN
        { \__regex_show_group_aux:nnnnN { ~(resetting) } }
      \cs_set_eq:NN \__regex_class:NnnnN \__regex_show_class:NnnnN
      \cs_set_protected:Npn \__regex_command_K:
        { \__regex_show_one:n { reset~match~start~(\iow_char:N\\K) } }
      \cs_set_protected:Npn \__regex_assertion:Nn ##1##2
        {
          \__regex_show_one:n
            { \bool_if:NF ##1 { negative~ } assertion:~##2 }
        }
      \cs_set:Npn \__regex_b_test: { word~boundary }
      \cs_set:Npn \__regex_Z_test: { anchor~at~end~(\iow_char:N\\Z) }
      \cs_set:Npn \__regex_A_test: { anchor~at~start~(\iow_char:N\\A) }
      \cs_set:Npn \__regex_G_test: { anchor~at~start~of~match~(\iow_char:N\\G) }
      \cs_set_protected:Npn \__regex_item_caseful_equal:n ##1
        { \__regex_show_one:n { char~code~\__regex_show_char:n{##1} } }
      \cs_set_protected:Npn \__regex_item_caseful_range:nn ##1##2
        {
          \__regex_show_one:n
            { range~[\__regex_show_char:n{##1}, \__regex_show_char:n{##2}] }
        }
      \cs_set_protected:Npn \__regex_item_caseless_equal:n ##1
        { \__regex_show_one:n { char~code~\__regex_show_char:n{##1}~(caseless) } }
      \cs_set_protected:Npn \__regex_item_caseless_range:nn ##1##2
        {
          \__regex_show_one:n
            { Range~[\__regex_show_char:n{##1}, \__regex_show_char:n{##2}]~(caseless) }
        }
      \cs_set_protected:Npn \__regex_item_catcode:nT
        { \__regex_show_item_catcode:NnT \c_true_bool }
      \cs_set_protected:Npn \__regex_item_catcode_reverse:nT
        { \__regex_show_item_catcode:NnT \c_false_bool }
      \cs_set_protected:Npn \__regex_item_reverse:n
        { \__regex_show_scope:nn { Reversed~match } }
      \cs_set_protected:Npn \__regex_item_exact:nn ##1##2
        { \__regex_show_one:n { char~\__regex_show_char:n{##2},~catcode~##1 } }
      \cs_set_eq:NN \__regex_item_exact_cs:n \__regex_show_item_exact_cs:n
      \cs_set_protected:Npn \__regex_item_cs:n
        { \__regex_show_scope:nn { control~sequence } }
      \cs_set:cpn { __regex_prop_.: } { \__regex_show_one:n { any~token } }
      \seq_clear:N \l__regex_show_prefix_seq
      \__regex_show_push:n { ~ }
      \cs_if_exist_use:N #1
      \tl_build_end:N \l__regex_build_tl
      \exp_args:NNNo
    \group_end:
    \tl_set:Nn \l__regex_internal_a_tl { \l__regex_build_tl }
  }
```

(*End of definition for* `\__regex_show:N`.)

`\__regex_show_char:n` Show a single character, together with its ascii representation if available. This could be extended to beyond ascii. It is not ideal for parentheses themselves.

```
\cs_new:Npn \__regex_show_char:n #1
  {
    \int_eval:n {#1}
    \int_compare:nT { 32 <= #1 <= 126 }
      { ~ ( \char_generate:nn {#1} {12} ) }
  }
```

(*End of definition for* `\__regex_show_char:n`.)

`\__regex_show_one:n`

Every part of the final message go through this function, which adds one line to the output, with the appropriate prefix.

```
\cs_new_protected:Npn \__regex_show_one:n #1
  {
    \int_incr:N \l__regex_show_lines_int
    \tl_build_put_right:Ne \l__regex_build_tl
      {
        \exp_not:N \iow_newline:
        \seq_map_function:NN \l__regex_show_prefix_seq \use:n
        #1
      }
  }
```

(*End of definition for* `\__regex_show_one:n`.)

`\__regex_show_push:n`
`\__regex_show_pop:`
`\__regex_show_scope:nn`

Enter and exit levels of nesting. The `scope` function prints its first argument as an "introduction", then performs its second argument in a deeper level of nesting.

```
\cs_new_protected:Npn \__regex_show_push:n #1
  { \seq_put_right:Ne \l__regex_show_prefix_seq { #1 ~ } }
\cs_new_protected:Npn \__regex_show_pop:
  { \seq_pop_right:NN \l__regex_show_prefix_seq \l__regex_internal_a_tl }
\cs_new_protected:Npn \__regex_show_scope:nn #1#2
  {
    \__regex_show_one:n {#1}
    \__regex_show_push:n { ~ }
    #2
    \__regex_show_pop:
  }
```

(*End of definition for* `\__regex_show_push:n` , `\__regex_show_pop:` , *and* `\__regex_show_scope:nn`.)

`\__regex_show_group_aux:nnnnN`

We display all groups in the same way, simply adding a message, `(no capture)` or `(resetting)`, to special groups. The odd `\use_ii:nn` avoids printing a spurious `+-branch` for the first branch.

```
\cs_new_protected:Npn \__regex_show_group_aux:nnnnN #1#2#3#4#5
  {
    \__regex_show_one:n { ,-group~begin #1 }
    \__regex_show_push:n { | }
    \use_ii:nn #2
    \__regex_show_pop:
    \__regex_show_one:n
      { `-group~end \__regex_msg_repeated:nnN {#3} {#4} #5 }
  }
```

(*End of definition for* `\__regex_show_group_aux:nnnnN`.)

__regex_show_class:NnnnN I'm entirely unhappy about this function: I couldn't find a way to test if a class is a single test. Instead, collect the representation of the tests in the class. If that had more than one line, write `Match` or `Don't match` on its own line, with the repeating information if any. Then the various tests on lines of their own, and finally a line. Otherwise, we need to evaluate the representation of the tests again (since the prefix is incorrect). That's clunky, but not too expensive, since it's only one test.

```
\cs_new:Npn \__regex_show_class:NnnnN #1#2#3#4#5
  {
    \group_begin:
      \tl_build_begin:N \l__regex_build_tl
      \int_zero:N \l__regex_show_lines_int
      \__regex_show_push:n {~}
      #2
    \int_compare:nTF { \l__regex_show_lines_int = \c_zero_int }
      {
        \group_end:
        \__regex_show_one:n { \bool_if:NTF #1 { Fail } { Pass } }
      }
      {
        \bool_if:nTF
          { #1 && \int_compare_p:n { \l__regex_show_lines_int = \c_one_int } }
          {
            \group_end:
            #2
            \tl_build_put_right:Nn \l__regex_build_tl
              { \__regex_msg_repeated:nnN {#3} {#4} #5 }
          }
          {
              \tl_build_end:N \l__regex_build_tl
              \exp_args:NNNo
            \group_end:
            \tl_set:Nn \l__regex_internal_a_tl \l__regex_build_tl
            \__regex_show_one:n
              {
                \bool_if:NTF #1 { Match } { Don't~match }
                \__regex_msg_repeated:nnN {#3} {#4} #5
              }
            \tl_build_put_right:Ne \l__regex_build_tl
              { \exp_not:o \l__regex_internal_a_tl }
          }
      }
  }
```

(*End of definition for* `\__regex_show_class:NnnnN`.)

__regex_show_item_catcode:NnT Produce a sequence of categories which the catcode bitmap `#2` contains, and show it, indenting the tests on which this catcode constraint applies.

```
\cs_new_protected:Npn \__regex_show_item_catcode:NnT #1#2
  {
    \seq_set_split:Nnn \l__regex_internal_seq { } { CBEMTPUDSLOA }
    \seq_set_filter:NNn \l__regex_internal_seq \l__regex_internal_seq
      { \int_if_odd_p:n { #2 / \int_use:c { c__regex_catcode_##1_int } } }
    \__regex_show_scope:nn
      {
```

```
        categories~
        \seq_map_function:NN \l__regex_internal_seq \use:n
        , ~
        \bool_if:NF #1 { negative~ } class
      }
  }
```

(*End of definition for* `\__regex_show_item_catcode:NnT`.)

`\__regex_show_item_exact_cs:n`

```
\cs_new_protected:Npn \__regex_show_item_exact_cs:n #1
  {
    \seq_set_split:Nnn \l__regex_internal_seq { \scan_stop: } {#1}
    \seq_set_map_e:NNn \l__regex_internal_seq
      \l__regex_internal_seq { \iow_char:N\\##1 }
    \__regex_show_one:n
      { control~sequence~ \seq_use:Nn \l__regex_internal_seq { ~or~ } }
  }
```

(*End of definition for* `\__regex_show_item_exact_cs:n`.)

46.4 Building

46.4.1 Variables used while building

`\l__regex_min_state_int`
`\l__regex_max_state_int`

The last state that was allocated is `\l__regex_max_state_int` -1, so that `\l__regex_max_state_int` always points to a free state. The `min_state` variable is 1 to begin with, but gets shifted in nested calls to the matching code, namely in `\c{...}` constructions.

```
\int_new:N  \l__regex_min_state_int
\int_set:Nn \l__regex_min_state_int { 1 }
\int_new:N  \l__regex_max_state_int
```

(*End of definition for* `\l__regex_min_state_int` *and* `\l__regex_max_state_int`.)

`\l__regex_left_state_int`
`\l__regex_right_state_int`
`\l__regex_left_state_seq`
`\l__regex_right_state_seq`

Alternatives are implemented by branching from a `left` state into the various choices, then merging those into a `right` state. We store information about those states in two sequences. Those states are also used to implement group quantifiers. Most often, the left and right pointers only differ by 1.

```
\int_new:N  \l__regex_left_state_int
\int_new:N  \l__regex_right_state_int
\seq_new:N  \l__regex_left_state_seq
\seq_new:N  \l__regex_right_state_seq
```

(*End of definition for* `\l__regex_left_state_int` *and others.*)

`\l__regex_capturing_group_int`

`\l__regex_capturing_group_int` is the next ID number to be assigned to a capturing group. This starts at 0 for the group enclosing the full regular expression, and groups are counted in the order of their left parenthesis, except when encountering `resetting` groups.

```
\int_new:N  \l__regex_capturing_group_int
```

(*End of definition for* `\l__regex_capturing_group_int`.)

46.4.2 Framework

This phase is about going from a compiled regex to an NFA. Each state of the NFA is stored in a `\toks`. The operations which can appear in the `\toks` are

- `\__regex_action_start_wildcard:N` ⟨*boolean*⟩ inserted at the start of the regular expression, where a `true` ⟨*boolean*⟩ makes it unanchored.
- `\__regex_action_success:` marks the exit state of the NFA.
- `\__regex_action_cost:n` {⟨*shift*⟩} is a transition from the current ⟨*state*⟩ to ⟨*state*⟩ + ⟨*shift*⟩, which consumes the current character: the target state is saved and will be considered again when matching at the next position.
- `\__regex_action_free:n` {⟨*shift*⟩}, and `\__regex_action_free_group:n` {⟨*shift*⟩} are free transitions, which immediately perform the actions for the state ⟨*state*⟩ + ⟨*shift*⟩ of the NFA. They differ in how they detect and avoid infinite loops. For now, we just need to know that the `group` variant must be used for transitions back to the start of a group.
- `\__regex_action_submatch:nN` {⟨*group*⟩} ⟨*key*⟩ where the ⟨*key*⟩ is `<` or `>` for the beginning or end of group numbered ⟨*group*⟩. This causes the current position in the query to be stored as the ⟨*key*⟩ submatch boundary.
- One of these actions, within a conditional.

We strive to preserve the following properties while building.

- The current capturing group is `capturing_group` − 1, and if a group opened now it would be labelled `capturing_group`.
- The last allocated state is `max_state` − 1, so `max_state` is a free state.
- The `left_state` points to a state to the left of the current group or of the last class.
- The `right_state` points to a newly created, empty state, with some transitions leading to it.
- The `left/right` sequences hold a list of the corresponding end-points of nested groups.

`\__regex_build:n`
`\__regex_build_aux:Nn`
`\__regex_build:N`
`\__regex_build_aux:NN`

The `n`-type function first compiles its argument. Reset some variables. Allocate two states, and put a wildcard in state 0 (transitions to state 1 and 0 state). Then build the regex within a (capturing) group numbered 0 (current value of `capturing_group`). Finally, if the match reaches the last state, it is successful. A `false` boolean for argument `#1` for the auxiliaries will suppress the wildcard and make the match anchored: used for `\peek_regex:nTF` and similar.

```
\cs_new_protected:Npn \__regex_build:n
  { \__regex_build_aux:Nn \c_true_bool }
\cs_new_protected:Npn \__regex_build:N
  { \__regex_build_aux:NN \c_true_bool }
\cs_new_protected:Npn \__regex_build_aux:Nn #1#2
  {
    \__regex_compile:n {#2}
```

```
    \__regex_build_aux:NN #1 \l__regex_internal_regex
  }
\cs_new_protected:Npn \__regex_build_aux:NN #1#2
  {
    \__regex_standard_escapechar:
    \int_zero:N \l__regex_capturing_group_int
    \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
    \__regex_build_new_state:
    \__regex_build_new_state:
    \__regex_toks_put_right:Nn \l__regex_left_state_int
      { \__regex_action_start_wildcard:N #1 }
    \__regex_group:nnnN {#2} { 1 } { 0 } \c_false_bool
    \__regex_toks_put_right:Nn \l__regex_right_state_int
      { \__regex_action_success: }
  }
```

(*End of definition for* `\__regex_build:n` *and others.*)

`\g__regex_case_int`

Case number that was successfully matched in `\regex_match_case:nn` and related functions.

```
\int_new:N \g__regex_case_int
```

(*End of definition for* `\g__regex_case_int`*.*)

`\l__regex_case_max_group_int`

The largest group number appearing in any of the ⟨*regex*⟩ in the argument of `\regex_match_case:nn` and related functions.

```
\int_new:N \l__regex_case_max_group_int
```

(*End of definition for* `\l__regex_case_max_group_int`*.*)

`\__regex_case_build:n`
`\__regex_case_build:e`
`\__regex_case_build_aux:Nn`
`\__regex_case_build_loop:n`

See `\__regex_build:n`, but with a loop.

```
\cs_new_protected:Npn \__regex_case_build:n #1
  {
    \__regex_case_build_aux:Nn \c_true_bool {#1}
    \int_gzero:N \g__regex_case_int
  }
\cs_generate_variant:Nn \__regex_case_build:n { e }
\cs_new_protected:Npn \__regex_case_build_aux:Nn #1#2
  {
    \__regex_standard_escapechar:
    \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
    \__regex_build_new_state:
    \__regex_build_new_state:
    \__regex_toks_put_right:Nn \l__regex_left_state_int
      { \__regex_action_start_wildcard:N #1 }
    %
    \__regex_build_new_state:
    \__regex_toks_put_left:Ne \l__regex_left_state_int
      { \__regex_action_submatch:nN \c_zero_int < }
    \__regex_push_lr_states:
    \int_zero:N \l__regex_case_max_group_int
    \int_gzero:N \g__regex_case_int
    \tl_map_inline:nn {#2}
      {
        \int_gincr:N \g__regex_case_int
```

```
      \__regex_case_build_loop:n {##1}
    }
  \int_set_eq:NN \l__regex_capturing_group_int \l__regex_case_max_group_int
  \__regex_pop_lr_states:
}
\cs_new_protected:Npn \__regex_case_build_loop:n #1
 {
  \int_set_eq:NN \l__regex_capturing_group_int \c_one_int
  \__regex_compile_use:n {#1}
  \int_set:Nn \l__regex_case_max_group_int
    { \int_max:nn \l__regex_case_max_group_int \l__regex_capturing_group_int }
  \seq_pop:NN \l__regex_right_state_seq \l__regex_internal_a_tl
  \int_set:Nn \l__regex_right_state_int \l__regex_internal_a_tl
  \__regex_toks_put_left:Ne \l__regex_right_state_int
    {
      \__regex_action_submatch:nN \c_zero_int >
      \int_gset:Nn \g__regex_case_int
        { \int_use:N \g__regex_case_int }
      \__regex_action_success:
    }
  \__regex_toks_clear:N \l__regex_max_state_int
  \seq_push:No \l__regex_right_state_seq
    { \int_use:N \l__regex_max_state_int }
  \int_incr:N \l__regex_max_state_int
 }
```

(*End of definition for* `\__regex_case_build:n`, `\__regex_case_build_aux:Nn`, *and* `\__regex_case_build_loop:n`.)

`\__regex_build_for_cs:n` The matching code relies on some global intarray variables, but only uses a range of their entries. Specifically,

- `\g__regex_state_active_intarray` from `\l__regex_min_state_int` to `\l__regex_max_state_int` − 1;

Here, in this nested call to the matching code, we need the new versions of this range to involve completely new entries of the intarray variables, so we begin by setting (the new) `\l__regex_min_state_int` to (the old) `\l__regex_max_state_int` to use higher entries.

When using a regex to match a cs, we don't insert a wildcard, we anchor at the end, and since we ignore submatches, there is no need to surround the expression with a group. However, for branches to work properly at the outer level, we need to put the appropriate `left` and `right` states in their sequence.

```
\cs_new_protected:Npn \__regex_build_for_cs:n #1
 {
  \int_set_eq:NN \l__regex_min_state_int \l__regex_max_state_int
  \__regex_build_new_state:
  \__regex_build_new_state:
  \__regex_push_lr_states:
  #1
  \__regex_pop_lr_states:
  \__regex_toks_put_right:Nn \l__regex_right_state_int
    {
      \if_int_compare:w -2 = \l__regex_curr_char_int
```

```
          \exp_after:wN \__regex_action_success:
        \fi:
      }
  }
```

(*End of definition for* `\__regex_build_for_cs:n`.)

46.4.3 Helpers for building an nfa

`\__regex_push_lr_states:`
`\__regex_pop_lr_states:`

When building the regular expression, we keep track of pointers to the left-end and right-end of each group without help from TeX's grouping.

```
\cs_new_protected:Npn \__regex_push_lr_states:
  {
    \seq_push:No \l__regex_left_state_seq
      { \int_use:N \l__regex_left_state_int }
    \seq_push:No \l__regex_right_state_seq
      { \int_use:N \l__regex_right_state_int }
  }
\cs_new_protected:Npn \__regex_pop_lr_states:
  {
    \seq_pop:NN \l__regex_left_state_seq  \l__regex_internal_a_tl
    \int_set:Nn \l__regex_left_state_int  \l__regex_internal_a_tl
    \seq_pop:NN \l__regex_right_state_seq \l__regex_internal_a_tl
    \int_set:Nn \l__regex_right_state_int \l__regex_internal_a_tl
  }
```

(*End of definition for* `\__regex_push_lr_states:` *and* `\__regex_pop_lr_states:`.)

`\__regex_build_transition_left:NNN`
`\__regex_build_transition_right:nNn`

Add a transition from `#2` to `#3` using the function `#1`. The `left` function is used for higher priority transitions, and the `right` function for lower priority transitions (which should be performed later). The signatures differ to reflect the differing usage later on. Both functions could be optimized.

```
\cs_new_protected:Npn \__regex_build_transition_left:NNN #1#2#3
  { \__regex_toks_put_left:Ne  #2 { #1 { \tex_the:D \__regex_int_eval:w #3 - #2 } } }
\cs_new_protected:Npn \__regex_build_transition_right:nNn #1#2#3
  { \__regex_toks_put_right:Ne #2 { #1 { \tex_the:D \__regex_int_eval:w #3 - #2 } } }
```

(*End of definition for* `\__regex_build_transition_left:NNN` *and* `\__regex_build_transition_right:nNn`.)

`\__regex_build_new_state:`

Add a new empty state to the NFA. Then update the `left`, `right`, and `max` states, so that the `right` state is the new empty state, and the `left` state points to the previously "current" state.

```
\cs_new_protected:Npn \__regex_build_new_state:
  {
    \__regex_toks_clear:N \l__regex_max_state_int
    \int_set_eq:NN \l__regex_left_state_int \l__regex_right_state_int
    \int_set_eq:NN \l__regex_right_state_int \l__regex_max_state_int
    \int_incr:N \l__regex_max_state_int
  }
```

(*End of definition for* `\__regex_build_new_state:`.)

__regex_build_transitions_laziness:NNNNN This function creates a new state, and puts two transitions starting from the old current state. The order of the transitions is controlled by #1, true for lazy quantifiers, and false for greedy quantifiers.

```
\cs_new_protected:Npn \__regex_build_transitions_laziness:NNNNN #1#2#3#4#5
  {
    \__regex_build_new_state:
    \__regex_toks_put_right:Ne \l__regex_left_state_int
      {
        \if_meaning:w \c_true_bool #1
          #2 { \tex_the:D \__regex_int_eval:w #3 - \l__regex_left_state_int }
          #4 { \tex_the:D \__regex_int_eval:w #5 - \l__regex_left_state_int }
        \else:
          #4 { \tex_the:D \__regex_int_eval:w #5 - \l__regex_left_state_int }
          #2 { \tex_the:D \__regex_int_eval:w #3 - \l__regex_left_state_int }
        \fi:
      }
  }
```

(*End of definition for* `\__regex_build_transitions_laziness:NNNNN`.)

46.4.4 Building classes

__regex_class:NnnnN
__regex_tests_action_cost:n

The arguments are: ⟨`boolean`⟩ {⟨*tests*⟩} {⟨*min*⟩} {⟨*more*⟩} ⟨`laziness`⟩. First store the tests with a trailing `\__regex_action_cost:n`, in the true branch of `\__regex_break_point:TF` for positive classes, or the false branch for negative classes. The integer ⟨`more`⟩ is 0 for fixed repetitions, -1 for unbounded repetitions, and ⟨`max`⟩ $-$ ⟨`min`⟩ for a range of repetitions.

```
\cs_new_protected:Npn \__regex_class:NnnnN #1#2#3#4#5
  {
    \cs_set:Npe \__regex_tests_action_cost:n ##1
      {
        \exp_not:n { \exp_not:n {#2} }
        \bool_if:NTF #1
          { \__regex_break_point:TF { \__regex_action_cost:n {##1} } { } }
          { \__regex_break_point:TF { } { \__regex_action_cost:n {##1} } }
      }
    \if_case:w - #4 \exp_stop_f:
           \__regex_class_repeat:n   {#3}
    \or:   \__regex_class_repeat:nN  {#3}      #5
    \else: \__regex_class_repeat:nnN {#3} {#4} #5
    \fi:
  }
\cs_new:Npn \__regex_tests_action_cost:n { \__regex_action_cost:n }
```

(*End of definition for* `\__regex_class:NnnnN` *and* `\__regex_tests_action_cost:n`.)

__regex_class_repeat:n This is used for a fixed number of repetitions. Build one state for each repetition, with a transition controlled by the tests that we have collected. That works just fine for #1 $= 0$ repetitions: nothing is built.

```
\cs_new_protected:Npn \__regex_class_repeat:n #1
  {
    \prg_replicate:nn {#1}
      {
```

```
      \__regex_build_new_state:
      \__regex_build_transition_right:nNn \__regex_tests_action_cost:n
        \l__regex_left_state_int \l__regex_right_state_int
    }
  }
```

(*End of definition for* `\__regex_class_repeat:n`.)

`\__regex_class_repeat:nN` This implements unbounded repetitions of a single class (*e.g.* the `*` and `+` quantifiers). If the minimum number `#1` of repetitions is 0, then build a transition from the current state to itself governed by the tests, and a free transition to a new state (hence skipping the tests). Otherwise, call `\__regex_class_repeat:n` for the code to match `#1` repetitions, and add free transitions from the last state to the previous one, and to a new one. In both cases, the order of transitions is controlled by the laziness boolean `#2`.

```
\cs_new_protected:Npn \__regex_class_repeat:nN #1#2
  {
    \if_int_compare:w #1 = \c_zero_int
      \__regex_build_transitions_laziness:NNNNN #2
        \__regex_action_free:n       \l__regex_right_state_int
        \__regex_tests_action_cost:n \l__regex_left_state_int
    \else:
      \__regex_class_repeat:n {#1}
      \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
      \__regex_build_transitions_laziness:NNNNN #2
        \__regex_action_free:n \l__regex_right_state_int
        \__regex_action_free:n \l__regex_internal_a_int
    \fi:
  }
```

(*End of definition for* `\__regex_class_repeat:nN`.)

`\__regex_class_repeat:nnN` We want to build the code to match from `#1` to `#1` + `#2` repetitions. Match `#1` repetitions (can be 0). Compute the final state of the next construction as `a`. Build `#2` >0 states, each with a transition to the next state governed by the tests, and a transition to the final state `a`. The computation of `a` is safe because states are allocated in order, starting from `max_state`.

```
\cs_new_protected:Npn \__regex_class_repeat:nnN #1#2#3
  {
    \__regex_class_repeat:n {#1}
    \int_set:Nn \l__regex_internal_a_int
      { \l__regex_max_state_int + #2 - \c_one_int }
    \prg_replicate:nn { #2 }
      {
        \__regex_build_transitions_laziness:NNNNN #3
          \__regex_action_free:n       \l__regex_internal_a_int
          \__regex_tests_action_cost:n \l__regex_right_state_int
      }
  }
```

(*End of definition for* `\__regex_class_repeat:nnN`.)

46.4.5 Building groups

__regex_group_aux:nnnnN

Arguments: {⟨*label*⟩} {⟨*contents*⟩} {⟨*min*⟩} {⟨*more*⟩} ⟨*laziness*⟩. If ⟨*min*⟩ is 0, we need to add a state before building the group, so that the thread which skips the group does not also set the start-point of the submatch. After adding one more state, the `left_state` is the left end of the group, from which all branches stem, and the `right_state` is the right end of the group, and all branches end their course in that state. We store those two integers to be queried for each branch, we build the NFA states for the contents `#2` of the group, and we forget about the two integers. Once this is done, perform the repetition: either exactly `#3` times, or `#3` or more times, or between `#3` and `#3` + `#4` times, with laziness `#5`. The ⟨*label*⟩ `#1` is used for submatch tracking. Each of the three auxiliaries expects `left_state` and `right_state` to be set properly.

```
\cs_new_protected:Npn \__regex_group_aux:nnnnN #1#2#3#4#5
  {
      \if_int_compare:w #3 = \c_zero_int
        \__regex_build_new_state:
        \__regex_build_transition_right:nNn \__regex_action_free_group:n
          \l__regex_left_state_int \l__regex_right_state_int
      \fi:
      \__regex_build_new_state:
      \__regex_push_lr_states:
      #2
      \__regex_pop_lr_states:
      \if_case:w - #4 \exp_stop_f:
             \__regex_group_repeat:nn   {#1} {#3}
      \or:   \__regex_group_repeat:nnN  {#1} {#3}      #5
      \else: \__regex_group_repeat:nnnN {#1} {#3} {#4} #5
      \fi:
  }
```

(*End of definition for* `\__regex_group_aux:nnnnN`.)

__regex_group:nnnN
__regex_group_no_capture:nnnN

Hand to `\__regex_group_aux:nnnnnN` the label of that group (expanded), and the group itself, with some extra commands to perform.

```
\cs_new_protected:Npn \__regex_group:nnnN #1
  {
    \exp_args:No \__regex_group_aux:nnnnN
      { \int_use:N \l__regex_capturing_group_int }
      {
        \int_incr:N \l__regex_capturing_group_int
        #1
      }
  }
\cs_new_protected:Npn \__regex_group_no_capture:nnnN
  { \__regex_group_aux:nnnnN { -1 } }
```

(*End of definition for* `\__regex_group:nnnN` *and* `\__regex_group_no_capture:nnnN`.)

__regex_group_resetting:nnnN
__regex_group_resetting_loop:nnNn

Again, hand the label -1 to `\__regex_group_aux:nnnnN`, but this time we work a little bit harder to keep track of the maximum group label at the end of any branch, and to reset the group number at each branch. This relies on the fact that a compiled regex always is a sequence of items of the form `\__regex_branch:n` {⟨*branch*⟩}.

```
\cs_new_protected:Npn \__regex_group_resetting:nnnN #1
```

```
  {
    \__regex_group_aux:nnnnN { -1 }
      {
        \exp_args:Noo \__regex_group_resetting_loop:nnNn
          { \int_use:N \l__regex_capturing_group_int }
          { \int_use:N \l__regex_capturing_group_int }
          #1
          { ?? \prg_break:n } { }
        \prg_break_point:
      }
  }
\cs_new_protected:Npn \__regex_group_resetting_loop:nnNn #1#2#3#4
  {
    \use_none:nn #3 { \int_set:Nn \l__regex_capturing_group_int {#1} }
    \int_set:Nn \l__regex_capturing_group_int {#2}
    #3 {#4}
    \exp_args:Ne \__regex_group_resetting_loop:nnNn
      { \int_max:nn {#1} \l__regex_capturing_group_int }
      {#2}
  }
```

(*End of definition for* `\__regex_group_resetting:nnnN` *and* `\__regex_group_resetting_loop:nnNn`.)

`\__regex_branch:n`

Add a free transition from the left state of the current group to a brand new state, starting point of this branch. Once the branch is built, add a transition from its last state to the right state of the group. The left and right states of the group are extracted from the relevant sequences.

```
\cs_new_protected:Npn \__regex_branch:n #1
  {
    \__regex_build_new_state:
    \seq_get:NN \l__regex_left_state_seq \l__regex_internal_a_tl
    \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
    \__regex_build_transition_right:nNn \__regex_action_free:n
      \l__regex_left_state_int \l__regex_right_state_int
    #1
    \seq_get:NN \l__regex_right_state_seq \l__regex_internal_a_tl
    \__regex_build_transition_right:nNn \__regex_action_free:n
      \l__regex_right_state_int \l__regex_internal_a_tl
  }
```

(*End of definition for* `\__regex_branch:n`.)

`\__regex_group_repeat:nn`

This function is called to repeat a group a fixed number of times `#2`; if this is 0 we remove the group altogether (but don't reset the `capturing_group` label). Otherwise, the auxiliary `\__regex_group_repeat_aux:n` copies `#2` times the `\toks` for the group, and leaves `internal_a` pointing to the left end of the last repetition. We only record the submatch information at the last repetition. Finally, add a state at the end (the transition to it has been taken care of by the replicating auxiliary).

```
\cs_new_protected:Npn \__regex_group_repeat:nn #1#2
  {
    \if_int_compare:w #2 = \c_zero_int
      \int_set:Nn \l__regex_max_state_int
        { \l__regex_left_state_int - \c_one_int }
      \__regex_build_new_state:
```

```
    \else:
      \__regex_group_repeat_aux:n {#2}
      \__regex_group_submatches:nNN {#1}
        \l__regex_internal_a_int \l__regex_right_state_int
      \__regex_build_new_state:
    \fi:
  }
```

(*End of definition for* `\__regex_group_repeat:nn`.)

`\__regex_group_submatches:nNN`

This inserts in states `#2` and `#3` the code for tracking submatches of the group `#1`, unless inhibited by a label of -1.

```
\cs_new_protected:Npn \__regex_group_submatches:nNN #1#2#3
  {
    \if_int_compare:w #1 > - \c_one_int
      \__regex_toks_put_left:Ne #2 { \__regex_action_submatch:nN {#1} < }
      \__regex_toks_put_left:Ne #3 { \__regex_action_submatch:nN {#1} > }
    \fi:
  }
```

(*End of definition for* `\__regex_group_submatches:nNN`.)

`\__regex_group_repeat_aux:n`

Here we repeat `\toks` ranging from `left_state` to `max_state`, `#1` > 0 times. First add a transition so that the copies "chain" properly. Compute the shift `c` between the original copy and the last copy we want. Shift the `right_state` and `max_state` to their final values. We then want to perform `c` copy operations. At the end, `b` is equal to the `max_state`, and `a` points to the left of the last copy of the group.

```
\cs_new_protected:Npn \__regex_group_repeat_aux:n #1
  {
    \__regex_build_transition_right:nNn \__regex_action_free:n
      \l__regex_right_state_int \l__regex_max_state_int
    \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
    \int_set_eq:NN \l__regex_internal_b_int \l__regex_max_state_int
    \if_int_compare:w \__regex_int_eval:w #1 > \c_one_int
      \int_set:Nn \l__regex_internal_c_int
        {
          ( #1 - \c_one_int )
          * ( \l__regex_internal_b_int - \l__regex_internal_a_int )
        }
      \int_add:Nn \l__regex_right_state_int \l__regex_internal_c_int
      \int_add:Nn \l__regex_max_state_int   \l__regex_internal_c_int
      \__regex_toks_memcpy:NNn
        \l__regex_internal_b_int
        \l__regex_internal_a_int
        \l__regex_internal_c_int
    \fi:
  }
```

(*End of definition for* `\__regex_group_repeat_aux:n`.)

`\__regex_group_repeat:nnN`

This function is called to repeat a group at least n times; the case $n = 0$ is very different from $n > 0$. Assume first that $n = 0$. Insert submatch tracking information at the start and end of the group, add a free transition from the right end to the "true" left state `a` (remember: in this case we had added an extra state before the left state). This forms the loop, which we break away from by adding a free transition from a to a new state.

Now consider the case $n > 0$. Repeat the group n times, chaining various copies with a free transition. Add submatch tracking only to the last copy, then add a free transition from the right end back to the left end of the last copy, either before or after the transition to move on towards the rest of the NFA. This transition can end up before submatch tracking, but that is irrelevant since it only does so when going again through the group, recording new matches. Finally, add a state; we already have a transition pointing to it from __regex_group_repeat_aux:n.

```
\cs_new_protected:Npn \__regex_group_repeat:nnN #1#2#3
  {
    \if_int_compare:w #2 = \c_zero_int
      \__regex_group_submatches:nNN {#1}
        \l__regex_left_state_int \l__regex_right_state_int
      \int_set:Nn \l__regex_internal_a_int
        { \l__regex_left_state_int - \c_one_int }
      \__regex_build_transition_right:nNn \__regex_action_free:n
        \l__regex_right_state_int \l__regex_internal_a_int
      \__regex_build_new_state:
      \if_meaning:w \c_true_bool #3
        \__regex_build_transition_left:NNN \__regex_action_free:n
          \l__regex_internal_a_int \l__regex_right_state_int
      \else:
        \__regex_build_transition_right:nNn \__regex_action_free:n
          \l__regex_internal_a_int \l__regex_right_state_int
      \fi:
    \else:
      \__regex_group_repeat_aux:n {#2}
      \__regex_group_submatches:nNN {#1}
        \l__regex_internal_a_int \l__regex_right_state_int
      \if_meaning:w \c_true_bool #3
        \__regex_build_transition_right:nNn \__regex_action_free_group:n
          \l__regex_right_state_int \l__regex_internal_a_int
      \else:
        \__regex_build_transition_left:NNN \__regex_action_free_group:n
          \l__regex_right_state_int \l__regex_internal_a_int
      \fi:
      \__regex_build_new_state:
    \fi:
  }
```

(*End of definition for* __regex_group_repeat:nnN.)

__regex_group_repeat:nnnN We wish to repeat the group between #2 and #2 + #3 times, with a laziness controlled by #4. We insert submatch tracking up front: in principle, we could avoid recording submatches for the first #2 copies of the group, but that forces us to treat specially the case #2 = 0. Repeat that group with submatch tracking #2 + #3 times (the maximum number of repetitions). Then our goal is to add #3 transitions from the end of the #2-th group, and each subsequent groups, to the end. For a lazy quantifier, we add those transitions to the left states, before submatch tracking. For the greedy case, we add the transitions to the right states, after submatch tracking and the transitions which go on with more repetitions. In the greedy case with #2 = 0, the transition which skips over all copies of the group must be added separately, because its starting state does not follow the normal pattern: we had to add it "by hand" earlier.

```
\cs_new_protected:Npn \__regex_group_repeat:nnnN #1#2#3#4
  {
    \__regex_group_submatches:nNN {#1}
      \l__regex_left_state_int \l__regex_right_state_int
    \__regex_group_repeat_aux:n { #2 + #3 }
    \if_meaning:w \c_true_bool #4
      \int_set_eq:NN \l__regex_left_state_int \l__regex_max_state_int
      \prg_replicate:nn { #3 }
        {
          \int_sub:Nn \l__regex_left_state_int
            { \l__regex_internal_b_int - \l__regex_internal_a_int }
          \__regex_build_transition_left:NNN \__regex_action_free:n
            \l__regex_left_state_int \l__regex_max_state_int
        }
    \else:
      \prg_replicate:nn { #3 - \c_one_int }
        {
          \int_sub:Nn \l__regex_right_state_int
            { \l__regex_internal_b_int - \l__regex_internal_a_int }
          \__regex_build_transition_right:nNn \__regex_action_free:n
            \l__regex_right_state_int \l__regex_max_state_int
        }
      \if_int_compare:w #2 = \c_zero_int
        \int_set:Nn \l__regex_right_state_int
          { \l__regex_left_state_int - \c_one_int }
      \else:
        \int_sub:Nn \l__regex_right_state_int
          { \l__regex_internal_b_int - \l__regex_internal_a_int }
      \fi:
      \__regex_build_transition_right:nNn \__regex_action_free:n
        \l__regex_right_state_int \l__regex_max_state_int
    \fi:
    \__regex_build_new_state:
  }
```

(*End of definition for* `\__regex_group_repeat:nnnN`.)

46.4.6 Others

__regex_assertion:Nn
__regex_b_test:
__regex_A_test:
__regex_G_test:
__regex_Z_test:

Usage: `\__regex_assertion:Nn` ⟨*boolean*⟩ {⟨*test*⟩}, where the ⟨*test*⟩ is either of the two other functions. Add a free transition to a new state, conditionally to the assertion test. The `\__regex_b_test:` test is used by the `\b` and `\B` escape: check if the last character was a word character or not, and do the same to the current character. The boundary-markers of the string are non-word characters for this purpose.

```
\cs_new_protected:Npn \__regex_assertion:Nn #1#2
  {
    \__regex_build_new_state:
    \__regex_toks_put_right:Ne \l__regex_left_state_int
      {
        \exp_not:n {#2}
        \__regex_break_point:TF
```

```
            \bool_if:NF #1 { { } }
            {
              \__regex_action_free:n
                {
                  \tex_the:D \__regex_int_eval:w
                    \l__regex_right_state_int - \l__regex_left_state_int
                }
            }
            \bool_if:NT #1 { { } }
        }
    }
\cs_new_protected:Npn \__regex_b_test:
  {
    \group_begin:
      \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_int
      \__regex_prop_w:
      \__regex_break_point:TF
        { \group_end: \__regex_item_reverse:n { \__regex_prop_w: } }
        { \group_end: \__regex_prop_w: }
  }
\cs_new_protected:Npn \__regex_Z_test:
  {
    \if_int_compare:w -2 = \l__regex_curr_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
\cs_new_protected:Npn \__regex_A_test:
  {
    \if_int_compare:w -2 = \l__regex_last_char_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
\cs_new_protected:Npn \__regex_G_test:
  {
    \if_int_compare:w \l__regex_curr_pos_int = \l__regex_start_pos_int
      \exp_after:wN \__regex_break_true:w
    \fi:
  }
```

(*End of definition for* `\__regex_assertion:Nn` *and others.*)

`\__regex_command_K:` Change the starting point of the 0-th submatch (full match), and transition to a new state, pretending that this is a fresh thread.

```
\cs_new_protected:Npn \__regex_command_K:
  {
    \__regex_build_new_state:
    \__regex_toks_put_right:Ne \l__regex_left_state_int
      {
        \__regex_action_submatch:nN \c_zero_int <
        \bool_set_true:N \l__regex_fresh_thread_bool
        \__regex_action_free:n
          {
            \tex_the:D \__regex_int_eval:w
              \l__regex_right_state_int - \l__regex_left_state_int
```

```
        }
      \bool_set_false:N \l__regex_fresh_thread_bool
    }
}
```

(*End of definition for* `\__regex_command_K:`.)

46.5 Matching

We search for matches by running all the execution threads through the NFA in parallel, reading one token of the query at each step. The NFA contains “free” transitions to other states, and transitions which “consume” the current token. For free transitions, the instruction at the new state of the NFA is performed immediately. When a transition consumes a character, the new state is appended to a list of “active states”, stored in `\g__regex_thread_info_intarray` (together with submatch information): this thread is made active again when the next token is read from the query. At every step (for each token in the query), we unpack that list of active states and the corresponding submatch props, and empty those.

If two paths through the NFA “collide” in the sense that they reach the same state after reading a given token, then they only differ in how they previously matched, and any future execution would be identical for both. (Note that this would be wrong in the presence of back-references.) Hence, we only need to keep one of the two threads: the thread with the highest priority. Our NFA is built in such a way that higher priority actions always come before lower priority actions, which makes things work.

The explanation in the previous paragraph may make us think that we simply need to keep track of which states were visited at a given step: after all, the loop generated when matching `(a?)*` against `a` is broken, isn’t it? No. The group first matches `a`, as it should, then repeats; it attempts to match `a` again but fails; it skips `a`, and finds out that this state has already been seen at this position in the query: the match stops. The capturing group is (wrongly) `a`. What went wrong is that a thread collided with itself, and the later version, which has gone through the group one more times with an empty match, should have a higher priority than not going through the group.

We solve this by distinguishing “normal” free transitions `\__regex_action_free:n` from transitions `\__regex_action_free_group:n` which go back to the start of the group. The former keeps threads unless they have been visited by a “completed” thread, while the latter kind of transition also prevents going back to a state visited by the current thread.

46.5.1 Variables used when matching

`\l__regex_min_pos_int`
`\l__regex_max_pos_int`
`\l__regex_curr_pos_int`
`\l__regex_start_pos_int`
`\l__regex_success_pos_int`

The tokens in the query are indexed from `min_pos` for the first to `max_pos` -1 for the last, and their information is stored in several arrays and `\toks` registers with those numbers. We match without backtracking, keeping all threads in lockstep at the `curr_pos` in the query. The starting point of the current match attempt is `start_pos`, and `success_pos`, updated whenever a thread succeeds, is used as the next starting position.

```
\int_new:N \l__regex_min_pos_int
\int_new:N \l__regex_max_pos_int
\int_new:N \l__regex_curr_pos_int
\int_new:N \l__regex_start_pos_int
\int_new:N \l__regex_success_pos_int
```

(*End of definition for* `\l__regex_min_pos_int` *and others.*)

`\l__regex_curr_char_int`
`\l__regex_curr_catcode_int`
`\l__regex_curr_token_tl`
`\l__regex_last_char_int`
`\l__regex_last_char_success_int`
`\l__regex_case_changed_char_int`

The character and category codes of the token at the current position and a token list expanding to that token; the character code of the token at the previous position; the character code of the token just before a successful match; and the character code of the result of changing the case of the current token (`A-Z`↔`a-z`). This last integer is only computed when necessary, and is otherwise `\c_max_int`. The `curr_char` variable is also used in various other phases to hold a character code.

```
\int_new:N \l__regex_curr_char_int
\int_new:N \l__regex_curr_catcode_int
\tl_new:N \l__regex_curr_token_tl
\int_new:N \l__regex_last_char_int
\int_new:N \l__regex_last_char_success_int
\int_new:N \l__regex_case_changed_char_int
```

(*End of definition for* `\l__regex_curr_char_int` *and others.*)

`\l__regex_curr_state_int`

For every character in the token list, each of the active states is considered in turn. The variable `\l__regex_curr_state_int` holds the state of the NFA which is currently considered: transitions are then given as shifts relative to the current state.

```
\int_new:N \l__regex_curr_state_int
```

(*End of definition for* `\l__regex_curr_state_int`*.*)

`\l__regex_curr_submatches_tl`
`\l__regex_success_submatches_tl`

The submatches for the thread which is currently active are stored in the `curr_submatches` list, which is almost a comma list, but ends with a comma. This list is stored by `\__regex_store_state:n` into an intarray variable, to be retrieved when matching at the next position. When a thread succeeds, this list is copied to `\l__regex_success_submatches_tl`: only the last successful thread remains there.

```
\tl_new:N \l__regex_curr_submatches_tl
\tl_new:N \l__regex_success_submatches_tl
```

(*End of definition for* `\l__regex_curr_submatches_tl` *and* `\l__regex_success_submatches_tl`*.*)

`\l__regex_step_int`

This integer, always even, is increased every time a character in the query is read, and not reset when doing multiple matches. We store in `\g__regex_state_active_intarray` the last step in which each ⟨*state*⟩ in the NFA was encountered. This lets us break infinite loops by not visiting the same state twice in the same step. In fact, the step we store is equal to `step` when we have started performing the operations of `\toks`⟨*state*⟩, but not finished yet. However, once we finish, we store `step` $+ 1$ in `\g__regex_state_active_intarray`. This is needed to track submatches properly (see building phase). The `step` is also used to attach each set of submatch information to a given iteration (and automatically discard it when it corresponds to a past step).

```
\int_new:N \l__regex_step_int
```

(*End of definition for* `\l__regex_step_int`*.*)

`\l__regex_min_thread_int`
`\l__regex_max_thread_int`

All the currently active threads are kept in order of precedence in `\g__regex_thread_info_intarray` together with the corresponding submatch information. Data in this intarray is organized as blocks from `min_thread` (included) to `max_thread` (excluded). At the start of every step, the whole array is unpacked, so that the space can immediately be reused, and `max_thread` is reset to `min_thread`, effectively clearing the array.

```
\int_new:N \l__regex_min_thread_int
\int_new:N \l__regex_max_thread_int
```

(*End of definition for* `\l__regex_min_thread_int` *and* `\l__regex_max_thread_int`*.*)

\g__regex_state_active_intarray
\g__regex_thread_info_intarray

`\g__regex_state_active_intarray` stores the last ⟨*step*⟩ in which each ⟨*state*⟩ was active. `\g__regex_thread_info_intarray` stores threads to be considered in the next step, more precisely the states in which these threads are.

```
\intarray_new:Nn \g__regex_state_active_intarray { 65536 }
\intarray_new:Nn \g__regex_thread_info_intarray { 65536 }
```

(*End of definition for* `\g__regex_state_active_intarray` *and* `\g__regex_thread_info_intarray`*.*)

\l__regex_matched_analysis_tl
\l__regex_curr_analysis_tl

The list `\l__regex_curr_analysis_tl` consists of a brace group containing three brace groups corresponding to the current token, with the same syntax as `\tl_analysis_map_inline:nn`. The list `\l__regex_matched_analysis_tl` (constructed under the `tl_build` machinery) has one item for each token that has already been treated so far in a given match attempt: each item consists of three brace groups with the same syntax as `\tl_analysis_map_inline:nn`.

```
\tl_new:N \l__regex_matched_analysis_tl
\tl_new:N \l__regex_curr_analysis_tl
```

(*End of definition for* `\l__regex_matched_analysis_tl` *and* `\l__regex_curr_analysis_tl`*.*)

\l__regex_every_match_tl

Every time a match is found, this token list is used. For single matching, the token list is empty. For multiple matching, the token list is set to repeat the matching, after performing some operation which depends on the user function. See `\__regex_single_match:` and `\__regex_multi_match:n`.

```
\tl_new:N \l__regex_every_match_tl
```

(*End of definition for* `\l__regex_every_match_tl`*.*)

\l__regex_fresh_thread_bool
\l__regex_empty_success_bool
__regex_if_two_empty_matches:F

When doing multiple matches, we need to avoid infinite loops where each iteration matches the same empty token list. When an empty token list is matched, the next successful match of the same empty token list is suppressed. We detect empty matches by setting `\l__regex_fresh_thread_bool` to `true` for threads which directly come from the start of the regex or from the `\K` command, and testing that boolean whenever a thread succeeds. The function `\__regex_if_two_empty_matches:F` is redefined at every match attempt, depending on whether the previous match was empty or not: if it was, then the function must cancel a purported success if it is empty and at the same spot as the previous match; otherwise, we definitely don't have two identical empty matches, so the function is `\use:n`.

```
\bool_new:N \l__regex_fresh_thread_bool
\bool_new:N \l__regex_empty_success_bool
\cs_new_eq:NN \__regex_if_two_empty_matches:F \use:n
```

(*End of definition for* `\l__regex_fresh_thread_bool`*,* `\l__regex_empty_success_bool`*, and* `\__regex_if_two_empty_matches:F`*.*)

\g__regex_success_bool
\l__regex_saved_success_bool
\l__regex_match_success_bool

The boolean `\l__regex_match_success_bool` is true if the current match attempt was successful, and `\g__regex_success_bool` is true if there was at least one successful match. This is the only global variable in this whole module, but we would need it to be local when matching a control sequence with `\c{...}`. This is done by saving the global variable into `\l__regex_saved_success_bool`, which is local, hence not affected by the changes due to inner regex functions.

```
\bool_new:N \g__regex_success_bool
\bool_new:N \l__regex_saved_success_bool
\bool_new:N \l__regex_match_success_bool
```

(*End of definition for* `\g__regex_success_bool`, `\l__regex_saved_success_bool`, *and* `\l__regex_match_success_bool`.)

46.5.2 Matching: framework

`\__regex_match:n`
`\__regex_match_cs:n`
`\__regex_match_init:`

Initialize the variables that should be set once for each user function (even for multiple matches). Namely, the overall matching is not yet successful; none of the states should be marked as visited (`\g__regex_state_active_intarray`), and we start at step 0; we pretend that there was a previous match ending at the start of the query, which was not empty (to avoid smothering an empty match at the start). Once all this is set up, we are ready for the ride. Find the first match.

```
\cs_new_protected:Npn \__regex_match:n #1
  {
    \__regex_match_init:
    \__regex_match_once_init:
    \tl_analysis_map_inline:nn {#1}
      { \__regex_match_one_token:nnN {##1} {##2} ##3 }
    \__regex_match_one_token:nnN { } { -2 } F
    \prg_break_point:Nn \__regex_maplike_break: { }
  }
\cs_new_protected:Npn \__regex_match_cs:n #1
  {
    \int_set_eq:NN \l__regex_min_thread_int \l__regex_max_thread_int
    \__regex_match_init:
    \__regex_match_once_init:
    \str_map_inline:nn {#1}
      {
        \tl_if_blank:nTF {##1}
          { \__regex_match_one_token:nnN {##1} {`##1} A }
          { \__regex_match_one_token:nnN {##1} {`##1} C }
      }
    \__regex_match_one_token:nnN { } { -2 } F
    \prg_break_point:Nn \__regex_maplike_break: { }
  }
\cs_new_protected:Npn \__regex_match_init:
  {
    \bool_gset_false:N \g__regex_success_bool
    \int_step_inline:nnn
      \l__regex_min_state_int { \l__regex_max_state_int - \c_one_int }
      {
        \__kernel_intarray_gset:Nnn
          \g__regex_state_active_intarray {##1} \c_one_int
      }
    \int_zero:N \l__regex_step_int
    \int_set:Nn \l__regex_min_pos_int { 2 }
    \int_set_eq:NN \l__regex_success_pos_int \l__regex_min_pos_int
    \int_set:Nn \l__regex_last_char_success_int { -2 }
    \tl_build_begin:N \l__regex_matched_analysis_tl
    \tl_clear:N \l__regex_curr_analysis_tl
    \int_set_eq:NN \l__regex_min_submatch_int \c_one_int
    \int_set_eq:NN \l__regex_submatch_int \l__regex_min_submatch_int
    \bool_set_false:N \l__regex_empty_success_bool
  }
```

(*End of definition for* `\__regex_match:n`*,* `\__regex_match_cs:n`*, and* `\__regex_match_init:`*.*)

`\__regex_match_once_init:`

This function resets various variables used when finding one match. It is called before the loop through characters, and every time we find a match, before searching for another match (this is controlled by the `every_match` token list).

First initialize some variables: set the conditional which detects identical empty matches; this match attempt starts at the previous `success_pos`, is not yet successful, and has no submatches yet; clear the array of active threads, and put the starting state 0 in it. We are then almost ready to read our first token in the query, but we actually start one position earlier than the start because `\__regex_match_one_token:nnN` increments `\l__regex_curr_pos_int` and saves `\l__regex_curr_char_int` as the `last_char` so that word boundaries can be correctly identified.

```
\cs_new_protected:Npn \__regex_match_once_init:
  {
    \if_meaning:w \c_true_bool \l__regex_empty_success_bool
      \cs_set:Npn \__regex_if_two_empty_matches:F
        {
          \int_compare:nNnF
            \l__regex_start_pos_int = \l__regex_curr_pos_int
        }
    \else:
      \cs_set_eq:NN \__regex_if_two_empty_matches:F \use:n
    \fi:
    \int_set_eq:NN \l__regex_start_pos_int \l__regex_success_pos_int
    \bool_set_false:N \l__regex_match_success_bool
    \tl_set:Ne \l__regex_curr_submatches_tl
      { \prg_replicate:nn { 2 * \l__regex_capturing_group_int } { 0 , } }
    \int_set_eq:NN \l__regex_max_thread_int \l__regex_min_thread_int
    \__regex_store_state:n { \l__regex_min_state_int }
    \int_set:Nn \l__regex_curr_pos_int { \l__regex_start_pos_int - \c_one_int }
    \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_success_int
    \tl_build_get_intermediate:NN \l__regex_matched_analysis_tl \l__regex_internal_a_tl
    \exp_args:NNf \__regex_match_once_init_aux:
    \tl_map_inline:nn
      { \exp_after:wN \l__regex_internal_a_tl \l__regex_curr_analysis_tl }
      { \__regex_match_one_token:nnN ##1 }
    \prg_break_point:Nn \__regex_maplike_break: { }
  }
\cs_new_protected:Npn \__regex_match_once_init_aux:
  {
    \tl_build_begin:N \l__regex_matched_analysis_tl
    \tl_clear:N \l__regex_curr_analysis_tl
  }
```

(*End of definition for* `\__regex_match_once_init:`*.*)

`\__regex_single_match:`
`\__regex_multi_match:n`

For a single match, the overall success is determined by whether the only match attempt is a success. When doing multiple matches, the overall matching is successful as soon as any match succeeds. Perform the action `#1`, then find the next match.

```
\cs_new_protected:Npn \__regex_single_match:
  {
    \tl_set:Nn \l__regex_every_match_tl
      {
```

```
      \bool_gset_eq:NN
        \g__regex_success_bool
        \l__regex_match_success_bool
      \__regex_maplike_break:
    }
  }
\cs_new_protected:Npn \__regex_multi_match:n #1
  {
    \tl_set:Nn \l__regex_every_match_tl
      {
        \if_meaning:w \c_false_bool \l__regex_match_success_bool
          \exp_after:wN \__regex_maplike_break:
        \fi:
        \bool_gset_true:N \g__regex_success_bool
        #1
        \__regex_match_once_init:
      }
  }
```

(*End of definition for* `\__regex_single_match:` *and* `\__regex_multi_match:n`.)

`\__regex_match_one_token:nnN`
`\__regex_match_one_active:n`

At each new position, set some variables and get the new character and category from the query. Then unpack the array of active threads, and clear it by resetting its length (`max_thread`). This results in a sequence of `\__regex_use_state_and_submatches:w` ⟨*state*⟩,⟨*submatch-clist*⟩; and we consider those states one by one in order. As soon as a thread succeeds, exit the step, and, if there are threads to consider at the next position, and we have not reached the end of the string, repeat the loop. Otherwise, the last thread that succeeded is the match. We explain the `fresh_thread` business when describing `\__regex_action_wildcard:`.

```
\cs_new_protected:Npn \__regex_match_one_token:nnN #1#2#3
  {
    \int_add:Nn \l__regex_step_int { 2 }
    \int_incr:N \l__regex_curr_pos_int
    \int_set_eq:NN \l__regex_last_char_int \l__regex_curr_char_int
    \cs_set_eq:NN \__regex_maybe_compute_ccc: \__regex_compute_case_changed_char:
    \tl_set:Nn \l__regex_curr_token_tl {#1}
    \int_set:Nn \l__regex_curr_char_int {#2}
    \int_set:Nn \l__regex_curr_catcode_int { "#3 }
    \tl_build_put_right:Ne \l__regex_matched_analysis_tl
      { \exp_not:o \l__regex_curr_analysis_tl }
    \tl_set:Nn \l__regex_curr_analysis_tl { { {#1} {#2} #3 } }
    \use:e
      {
        \int_set_eq:NN \l__regex_max_thread_int \l__regex_min_thread_int
        \int_step_function:nnN
          \l__regex_min_thread_int
          { \l__regex_max_thread_int - \c_one_int }
          \__regex_match_one_active:n
      }
    \prg_break_point:
    \bool_set_false:N \l__regex_fresh_thread_bool
    \if_int_compare:w \l__regex_max_thread_int > \l__regex_min_thread_int
      \if_int_compare:w -2 < \l__regex_curr_char_int
        \exp_after:wN \use_i:nn
```

```
      \fi:
    \fi:
    \l__regex_every_match_tl
  }
\cs_new:Npn \__regex_match_one_active:n #1
  {
    \__regex_use_state_and_submatches:w
    \__kernel_intarray_range_to_clist:Nnn
      \g__regex_thread_info_intarray
      { \c_one_int + #1 * (\l__regex_capturing_group_int * 2 + \c_one_int) }
      { (\c_one_int + #1) * (\l__regex_capturing_group_int * 2 + \c_one_int) }
    ;
  }
```

(*End of definition for* `\__regex_match_one_token:nnN` *and* `\__regex_match_one_active:n`.)

46.5.3 Using states of the nfa

`\__regex_use_state:`

Use the current NFA instruction. The state is initially marked as belonging to the current `step`: this allows normal free transition to repeat, but group-repeating transitions won't. Once we are done exploring all the branches it spawned, the state is marked as `step` $+1$: any thread hitting it at that point will be terminated.

```
\cs_new_protected:Npn \__regex_use_state:
  {
    \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
      \l__regex_curr_state_int \l__regex_step_int
    \__regex_toks_use:w \l__regex_curr_state_int
    \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
      \l__regex_curr_state_int
      { \__regex_int_eval:w \l__regex_step_int + \c_one_int \scan_stop: }
  }
```

(*End of definition for* `\__regex_use_state:`.)

`\__regex_use_state_and_submatches:w`

This function is called as one item in the array of active threads after that array has been unpacked for a new step. Update the `curr_state` and `curr_submatches` and use the state if it has not yet been encountered at this step.

```
\cs_new_protected:Npn \__regex_use_state_and_submatches:w #1 , #2 ;
  {
    \int_set:Nn \l__regex_curr_state_int {#1}
    \if_int_compare:w
        \__kernel_intarray_item:Nn \g__regex_state_active_intarray
          \l__regex_curr_state_int
                      < \l__regex_step_int
      \tl_set:Nn \l__regex_curr_submatches_tl { #2 , }
      \exp_after:wN \__regex_use_state:
    \fi:
    \scan_stop:
  }
```

(*End of definition for* `\__regex_use_state_and_submatches:w`.)

46.5.4 Actions when matching

__regex_action_start_wildcard:N

For an unanchored match, state 0 has a free transition to the next and a costly one to itself, to repeat at the next position. To catch repeated identical empty matches, we need to know if a successful thread corresponds to an empty match. The instruction resetting `\l__regex_fresh_thread_bool` may be skipped by a successful thread, hence we had to add it to `\__regex_match_one_token:nnN` too.

```
\cs_new_protected:Npn \__regex_action_start_wildcard:N #1
  {
    \bool_set_true:N \l__regex_fresh_thread_bool
    \__regex_action_free:n {1}
    \bool_set_false:N \l__regex_fresh_thread_bool
    \bool_if:NT #1 { \__regex_action_cost:n {0} }
  }
```

(*End of definition for* `\__regex_action_start_wildcard:N`.)

__regex_action_free:n
__regex_action_free_group:n
__regex_action_free_aux:nn

These functions copy a thread after checking that the NFA state has not already been used at this position. If not, store submatches in the new state, and insert the instructions for that state in the input stream. Then restore the old value of `\l__regex_curr_state_int` and of the current submatches. The two types of free transitions differ by how they test that the state has not been encountered yet: the `group` version is stricter, and will not use a state if it was used earlier in the current thread, hence forcefully breaking the loop, while the "normal" version will revisit a state even within the thread itself.

```
\cs_new_protected:Npn \__regex_action_free:n
  { \__regex_action_free_aux:nn { > \l__regex_step_int \else: } }
\cs_new_protected:Npn \__regex_action_free_group:n
  { \__regex_action_free_aux:nn { < \l__regex_step_int } }
\cs_new_protected:Npn \__regex_action_free_aux:nn #1#2
  {
    \use:e
      {
        \int_add:Nn \l__regex_curr_state_int {#2}
        \exp_not:n
          {
            \if_int_compare:w
                \__kernel_intarray_item:Nn \g__regex_state_active_intarray
                  \l__regex_curr_state_int
                #1
              \exp_after:wN \__regex_use_state:
            \fi:
          }
        \int_set:Nn \l__regex_curr_state_int
          { \int_use:N \l__regex_curr_state_int }
        \tl_set:Nn \exp_not:N \l__regex_curr_submatches_tl
          { \exp_not:o \l__regex_curr_submatches_tl }
      }
  }
```

(*End of definition for* `\__regex_action_free:n`*,* `\__regex_action_free_group:n`*, and* `\__regex_action_free_aux:nn`.)

__regex_action_cost:n

A transition which consumes the current character and shifts the state by `#1`. The resulting state is stored in the appropriate array for use at the next position, and we also store the current submatches.

```
\cs_new_protected:Npn \__regex_action_cost:n #1
  {
    \exp_args:No \__regex_store_state:n
      { \tex_the:D \__regex_int_eval:w \l__regex_curr_state_int + #1 }
  }
```

(*End of definition for* `\__regex_action_cost:n`.)

`\__regex_store_state:n` `\__regex_store_submatches:` Put the given state and current submatch information in `\g__regex_thread_info_intarray`, and increment the length of the array.

```
\cs_new_protected:Npn \__regex_store_state:n #1
  {
    \exp_args:No \__regex_store_submatches:nn
      \l__regex_curr_submatches_tl {#1}
    \int_incr:N \l__regex_max_thread_int
  }
\cs_new_protected:Npn \__regex_store_submatches:nn #1#2
  {
    \__kernel_intarray_gset_range_from_clist:Nnn
      \g__regex_thread_info_intarray
      {
        \__regex_int_eval:w
        \c_one_int + \l__regex_max_thread_int *
        (\l__regex_capturing_group_int * 2 + \c_one_int)
      }
      { #2 , #1 }
  }
```

(*End of definition for* `\__regex_store_state:n` *and* `\__regex_store_submatches:`.)

`\__regex_disable_submatches:` Some user functions don't require tracking submatches. We get a performance improvement by simply defining the relevant functions to remove their argument and do nothing with it.

```
\cs_new_protected:Npn \__regex_disable_submatches:
  {
    \cs_set_protected:Npn \__regex_store_submatches:n ##1 { }
    \cs_set_protected:Npn \__regex_action_submatch:nN ##1##2 { }
  }
```

(*End of definition for* `\__regex_disable_submatches:`.)

`\__regex_action_submatch:nN` `\__regex_action_submatch_aux:w` `\__regex_action_submatch_auxii:w` `\__regex_action_submatch_auxiii:w` `\__regex_action_submatch_auxiv:w` Update the current submatches with the information from the current position. Maybe a bottleneck.

```
\cs_new_protected:Npn \__regex_action_submatch:nN #1#2
  {
    \exp_after:wN \__regex_action_submatch_aux:w
    \l__regex_curr_submatches_tl ; {#1} #2
  }
\cs_new_protected:Npn \__regex_action_submatch_aux:w #1 ; #2#3
  {
    \tl_set:Ne \l__regex_curr_submatches_tl
      {
        \prg_replicate:nn
          { #2 \if_meaning:w > #3 + \l__regex_capturing_group_int \fi: }
```

```
        { \__regex_action_submatch_auxii:w }
      \__regex_action_submatch_auxiii:w
      #1
    }
  }
\cs_new:Npn \__regex_action_submatch_auxii:w
    #1 \__regex_action_submatch_auxiii:w #2 ,
  { #2 , #1 \__regex_action_submatch_auxiii:w }
\cs_new:Npn \__regex_action_submatch_auxiii:w #1 ,
  { \int_use:N \l__regex_curr_pos_int , }
```

(*End of definition for* `\__regex_action_submatch:nN` *and others.*)

`\__regex_action_success:`

There is a successful match when an execution path reaches the last state in the NFA, unless this marks a second identical empty match. Then mark that there was a successful match; it is empty if it is "fresh"; and we store the current position and submatches. The current step is then interrupted with `\prg_break:`, and only paths with higher precedence are pursued further. The values stored here may be overwritten by a later success of a path with higher precedence.

```
\cs_new_protected:Npn \__regex_action_success:
  {
    \__regex_if_two_empty_matches:F
      {
        \bool_set_true:N \l__regex_match_success_bool
        \bool_set_eq:NN \l__regex_empty_success_bool
          \l__regex_fresh_thread_bool
        \int_set_eq:NN \l__regex_success_pos_int \l__regex_curr_pos_int
        \int_set_eq:NN \l__regex_last_char_success_int \l__regex_last_char_int
        \tl_build_begin:N \l__regex_matched_analysis_tl
        \tl_set_eq:NN \l__regex_success_submatches_tl
          \l__regex_curr_submatches_tl
        \prg_break:
      }
  }
```

(*End of definition for* `\__regex_action_success:`.)

46.6 Replacement

46.6.1 Variables and helpers used in replacement

`\l__regex_replacement_csnames_int`

The behaviour of closing braces inside a replacement text depends on whether a sequences `\c{` or `\u{` has been encountered. The number of "open" such sequences that should be closed by `}` is stored in `\l__regex_replacement_csnames_int`, and decreased by 1 by each `}`.

```
\int_new:N \l__regex_replacement_csnames_int
```

(*End of definition for* `\l__regex_replacement_csnames_int`.)

`\l__regex_replacement_category_tl`
`\l__regex_replacement_category_seq`

This sequence of letters is used to correctly restore categories in nested constructions such as `\cL(abc\cD(_)d)`.

```
\tl_new:N \l__regex_replacement_category_tl
\seq_new:N \l__regex_replacement_category_seq
```

(*End of definition for* \l__regex_replacement_category_tl *and* \l__regex_replacement_category_seq.)

\g__regex_balance_tl

This token list holds the replacement text for __regex_replacement_balance_one_match:n while it is being built incrementally.

```
\tl_new:N \g__regex_balance_tl
```

(*End of definition for* \g__regex_balance_tl.)

__regex_replacement_balance_one_match:n

This expects as an argument the first index of a set of entries in \g__regex_submatch_begin_intarray (and related arrays) which hold the submatch information for a given match. It can be used within an integer expression to obtain the brace balance incurred by performing the replacement on that match. This combines the braces lost by removing the match, braces added by all the submatches appearing in the replacement, and braces appearing explicitly in the replacement. Even though it is always redefined before use, we initialize it as for an empty replacement. An important property is that concatenating several calls to that function must result in a valid integer expression (hence a leading + in the actual definition).

```
\cs_new:Npn \__regex_replacement_balance_one_match:n #1
  { - \__regex_submatch_balance:n {#1} }
```

(*End of definition for* __regex_replacement_balance_one_match:n.)

__regex_replacement_do_one_match:n

The input is the same as __regex_replacement_balance_one_match:n. This function is redefined to expand to the part of the token list from the end of the previous match to a given match, followed by the replacement text. Hence concatenating the result of this function with all possible arguments (one call for each match), as well as the range from the end of the last match to the end of the string, produces the fully replaced token list. The initialization does not matter, but (as an example) we set it as for an empty replacement.

```
\cs_new:Npn \__regex_replacement_do_one_match:n #1
  {
    \__regex_query_range:nn
      { \__kernel_intarray_item:Nn \g__regex_submatch_prev_intarray {#1} }
      { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
  }
```

(*End of definition for* __regex_replacement_do_one_match:n.)

__regex_replacement_exp_not:N

This function lets us navigate around the fact that the primitive \exp_not:n requires a braced argument. As far as I can tell, it is only needed if the user tries to include in the replacement text a control sequence set equal to a macro parameter character, such as \c_parameter_token. Indeed, within an e/x-expanding assignment, \exp_not:N # behaves as a single #, whereas \exp_not:n {#} behaves as a doubled ##.

```
\cs_new:Npn \__regex_replacement_exp_not:N #1 { \exp_not:n {#1} }
```

(*End of definition for* __regex_replacement_exp_not:N.)

__regex_replacement_exp_not:V

This is used for the implementation of \u, and it gets redefined for \peek_regex_replace_once:nnTF.

```
\cs_new_eq:NN \__regex_replacement_exp_not:V \exp_not:V
```

(*End of definition for* __regex_replacement_exp_not:V.)

46.6.2 Query and brace balance

__regex_query_range:nn
__regex_query_range_loop:ww

When it is time to extract submatches from the token list, the various tokens are stored in `\toks` registers numbered from `\l__regex_min_pos_int` inclusive to `\l__regex_max_pos_int` exclusive. The function `\__regex_query_range:nn {`⟨*min*⟩`} {`⟨*max*⟩`}` unpacks registers from the position ⟨*min*⟩ to the position ⟨*max*⟩-1 included. Once this is expanded, a second `e`-expansion results in the actual tokens from the query. That second expansion is only done by user functions at the very end of their operation, after checking (and correcting) the brace balance first.

```
\cs_new:Npn \__regex_query_range:nn #1#2
  {
    \exp_after:wN \__regex_query_range_loop:ww
    \int_value:w \__regex_int_eval:w #1 \exp_after:wN ;
    \int_value:w \__regex_int_eval:w #2 ;
    \prg_break_point:
  }
\cs_new:Npn \__regex_query_range_loop:ww #1 ; #2 ;
  {
    \if_int_compare:w #1 < #2 \exp_stop_f:
    \else:
      \prg_break:n
    \fi:
    \__regex_toks_use:w #1 \exp_stop_f:
    \exp_after:wN \__regex_query_range_loop:ww
      \int_value:w \__regex_int_eval:w #1 + \c_one_int ; #2 ;
  }
```

(*End of definition for* `\__regex_query_range:nn` *and* `\__regex_query_range_loop:ww`*.*)

__regex_query_submatch:n

Find the start and end positions for a given submatch (of a given match).

```
\cs_new:Npn \__regex_query_submatch:n #1
  {
    \__regex_query_range:nn
      { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
      { \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray {#1} }
  }
```

(*End of definition for* `\__regex_query_submatch:n`*.*)

__regex_submatch_balance:n

Every user function must result in a balanced token list (unbalanced token lists cannot be stored by TeX). When we unpacked the query, we kept track of the brace balance, hence the contribution from a given range is the difference between the brace balances at the ⟨*max pos*⟩ and ⟨*min pos*⟩. These two positions are found in the corresponding "submatch" arrays.

```
\cs_new_protected:Npn \__regex_submatch_balance:n #1
  {
    \tex_the:D \__regex_int_eval:w
      \__regex_intarray_item:NnF \g__regex_balance_intarray
        {
          \__kernel_intarray_item:Nn
            \g__regex_submatch_end_intarray {#1}
        }
        \c_zero_int
      -
```

```
      \__regex_intarray_item:NnF \g__regex_balance_intarray
        {
          \__kernel_intarray_item:Nn
            \g__regex_submatch_begin_intarray {#1}
        }
        \c_zero_int
    \scan_stop:
  }
```

(*End of definition for* `\__regex_submatch_balance:n`.)

46.6.3 Framework

`\__regex_replacement:n`
`\__regex_replacement:e`
`\__regex_replacement_apply:Nn`
`\__regex_replacement_set:n`

The replacement text is built incrementally. We keep track in `\l__regex_balance_int` of the balance of explicit begin- and end-group tokens and we store in `\g__regex_balance_tl` some code to compute the brace balance from submatches (see its description). Detect unescaped right braces, and escaped characters, with trailing `\prg_do_nothing:` because some of the later function look-ahead. Once the whole replacement text has been parsed, make sure that there is no open csname. Finally, define the `balance_one_match` and `do_one_match` functions.

```
\cs_new_protected:Npn \__regex_replacement:n
  { \__regex_replacement_apply:Nn \__regex_replacement_set:n }
\cs_new_protected:Npn \__regex_replacement_apply:Nn #1#2
  {
    \group_begin:
      \tl_build_begin:N \l__regex_build_tl
      \int_zero:N \l__regex_balance_int
      \tl_gclear:N \g__regex_balance_tl
      \__regex_escape_use:nnnn
        {
          \if_charcode:w \c_right_brace_str ##1
            \__regex_replacement_rbrace:N
          \else:
            \if_charcode:w \c_left_brace_str ##1
              \__regex_replacement_lbrace:N
            \else:
              \__regex_replacement_normal:n
            \fi:
          \fi:
          ##1
        }
        { \__regex_replacement_escaped:N ##1 }
        { \__regex_replacement_normal:n ##1 }
        {#2}
      \prg_do_nothing: \prg_do_nothing:
      \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
        \msg_error:nne { regex } { replacement-missing-rbrace }
          { \int_use:N \l__regex_replacement_csnames_int }
        \tl_build_put_right:Ne \l__regex_build_tl
          { \prg_replicate:nn \l__regex_replacement_csnames_int \cs_end: }
      \fi:
      \seq_if_empty:NF \l__regex_replacement_category_seq
        {
```

```
        \msg_error:nne { regex } { replacement-missing-rparen }
          { \seq_count:N \l__regex_replacement_category_seq }
        \seq_clear:N \l__regex_replacement_category_seq
      }
    \tl_gput_right:Ne \g__regex_balance_tl
      { + \int_use:N \l__regex_balance_int }
    \tl_build_end:N \l__regex_build_tl
    \exp_args:NNo
  \group_end:
  #1 \l__regex_build_tl
}
\cs_generate_variant:Nn \__regex_replacement:n { e }
\cs_new_protected:Npn \__regex_replacement_set:n #1
  {
    \cs_set:Npn \__regex_replacement_do_one_match:n ##1
      {
        \__regex_query_range:nn
          {
            \__kernel_intarray_item:Nn
              \g__regex_submatch_prev_intarray {##1}
          }
          {
            \__kernel_intarray_item:Nn
              \g__regex_submatch_begin_intarray {##1}
          }
        #1
      }
    \exp_args:Nno \use:n
      { \cs_gset:Npn \__regex_replacement_balance_one_match:n ##1 }
      {
        \g__regex_balance_tl
        - \__regex_submatch_balance:n {##1}
      }
  }
```

(*End of definition for* `\__regex_replacement:n`*,* `\__regex_replacement_apply:Nn`*, and* `\__regex_replacement_set:n`*.*)

`\__regex_case_replacement:n`
`\__regex_case_replacement:e`

```
\tl_new:N \g__regex_case_replacement_tl
\tl_new:N \g__regex_case_balance_tl
\cs_new_protected:Npn \__regex_case_replacement:n #1
  {
    \tl_gset:Nn \g__regex_case_balance_tl
      {
        \if_case:w
          \__kernel_intarray_item:Nn
            \g__regex_submatch_case_intarray {##1}
      }
    \tl_gset_eq:NN \g__regex_case_replacement_tl \g__regex_case_balance_tl
    \tl_map_tokens:nn {#1}
      { \__regex_replacement_apply:Nn \__regex_case_replacement_aux:n }
    \tl_gset:No \g__regex_balance_tl
      { \g__regex_case_balance_tl \fi: }
```

```
    \exp_args:No \__regex_replacement_set:n
      { \g__regex_case_replacement_tl \fi: }
  }
\cs_generate_variant:Nn \__regex_case_replacement:n { e }
\cs_new_protected:Npn \__regex_case_replacement_aux:n #1
  {
    \tl_gput_right:Nn \g__regex_case_replacement_tl { \or: #1 }
    \tl_gput_right:No \g__regex_case_balance_tl
      { \exp_after:wN \or: \g__regex_balance_tl }
  }
```

(*End of definition for* `\__regex_case_replacement:n`.)

`\__regex_replacement_put:n`

This gets redefined for `\peek_regex_replace_once:nnTF`.

```
\cs_new_protected:Npn \__regex_replacement_put:n
  { \tl_build_put_right:Nn \l__regex_build_tl }
```

(*End of definition for* `\__regex_replacement_put:n`.)

`\__regex_replacement_normal:n`
`\__regex_replacement_normal_aux:N`

Most characters are simply sent to the output by `\tl_build_put_right:Nn`, unless a particular category code has been requested: then `\__regex_replacement_c_A:w` or a similar auxiliary is called. One exception is right parentheses, which restore the category code in place before the group started. Note that the sequence is non-empty there: it contains an empty entry corresponding to the initial value of `\l__regex_replacement_category_tl`. The argument `#1` is a single character (including the case of a catcode-other space). In case no specific catcode is requested, we take into account the current catcode regime (at the time the replacement is performed) as much as reasonable, with all impossible catcodes (escape, newline, etc.) being mapped to "other".

```
\cs_new_protected:Npn \__regex_replacement_normal:n #1
  {
    \int_compare:nNnTF \l__regex_replacement_csnames_int > \c_zero_int
      { \exp_args:No \__regex_replacement_put:n { \token_to_str:N #1 } }
      {
        \tl_if_empty:NTF \l__regex_replacement_category_tl
          { \__regex_replacement_normal_aux:N #1 }
          { % (
            \token_if_eq_charcode:NNTF #1 )
              {
                \seq_pop:NN \l__regex_replacement_category_seq
                  \l__regex_replacement_category_tl
              }
              {
                \use:c { __regex_replacement_c_ \l__regex_replacement_category_tl :w }
                ? #1
              }
          }
      }
  }
\cs_new_protected:Npn \__regex_replacement_normal_aux:N #1
  {
    \token_if_eq_charcode:NNTF #1 \c_space_token
      { \__regex_replacement_c_S:w }
      {
        \exp_after:wN \exp_after:wN
```

```
        \if_case:w \tex_catcode:D `#1 \exp_stop_f:
             \__regex_replacement_c_O:w
        \or: \__regex_replacement_c_B:w
        \or: \__regex_replacement_c_E:w
        \or: \__regex_replacement_c_M:w
        \or: \__regex_replacement_c_T:w
        \or: \__regex_replacement_c_O:w
        \or: \__regex_replacement_c_P:w
        \or: \__regex_replacement_c_U:w
        \or: \__regex_replacement_c_D:w
        \or: \__regex_replacement_c_O:w
        \or: \__regex_replacement_c_S:w
        \or: \__regex_replacement_c_L:w
        \or: \__regex_replacement_c_O:w
        \or: \__regex_replacement_c_A:w
        \else: \__regex_replacement_c_O:w
        \fi:
      }
    ? #1
  }
```

(*End of definition for* `\__regex_replacement_normal:n` *and* `\__regex_replacement_normal_aux:N`.)

__regex_replacement_escaped:N

As in parsing a regular expression, we use an auxiliary built from `#1` if defined. Otherwise, check for escaped digits (standing from submatches from 0 to 9): anything else is a raw character.

```
\cs_new_protected:Npn \__regex_replacement_escaped:N #1
  {
    \cs_if_exist_use:cF { __regex_replacement_#1:w }
      {
        \if_int_compare:w \c_one_int < 1#1 \exp_stop_f:
          \__regex_replacement_put_submatch:n {#1}
        \else:
          \__regex_replacement_normal:n {#1}
        \fi:
      }
  }
```

(*End of definition for* `\__regex_replacement_escaped:N`.)

46.6.4 Submatches

__regex_replacement_put_submatch:n
__regex_replacement_put_submatch_aux:n

Insert a submatch in the replacement text. This is dropped if the submatch number is larger than the number of capturing groups. Unless the submatch appears inside a `\c{...}` or `\u{...}` construction, it must be taken into account in the brace balance. Later on, `##1` will be replaced by a pointer to the 0-th submatch for a given match.

```
\cs_new_protected:Npn \__regex_replacement_put_submatch:n #1
  {
    \if_int_compare:w #1 < \l__regex_capturing_group_int
      \__regex_replacement_put_submatch_aux:n {#1}
    \else:
      \msg_expandable_error:nnff { regex } { submatch-too-big }
        {#1} { \int_eval:n { \l__regex_capturing_group_int - \c_one_int } }
    \fi:
```

```
    }
\cs_new_protected:Npn \__regex_replacement_put_submatch_aux:n #1
  {
    \tl_build_put_right:Nn \l__regex_build_tl
      { \__regex_query_submatch:n { \__regex_int_eval:w #1 + ##1 \scan_stop: } }
    \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
      \tl_gput_right:Nn \g__regex_balance_tl
        { + \__regex_submatch_balance:n { \__regex_int_eval:w #1 + ##1 \scan_stop: } }
    \fi:
  }
```

(*End of definition for* `\__regex_replacement_put_submatch:n` *and* `\__regex_replacement_put_submatch_aux:n`.)

`\__regex_replacement_g:w`
`\__regex_replacement_g_digits:NN`

Grab digits for the `\g` escape sequence in a primitive assignment to the integer `\l__regex_internal_a_int`. At the end of the run of digits, check that it ends with a right brace.

```
\cs_new_protected:Npn \__regex_replacement_g:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
      { \l__regex_internal_a_int = \__regex_replacement_g_digits:NN }
      { \__regex_replacement_error:NNN g #1 #2 }
  }
\cs_new:Npn \__regex_replacement_g_digits:NN #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
      {
        \if_int_compare:w \c_one_int < 1#2 \exp_stop_f:
          #2
          \exp_after:wN \use_i:nnn
          \exp_after:wN \__regex_replacement_g_digits:NN
        \else:
          \exp_stop_f:
          \exp_after:wN \__regex_replacement_error:NNN
          \exp_after:wN g
        \fi:
      }
      {
        \exp_stop_f:
        \if_meaning:w \__regex_replacement_rbrace:N #1
          \exp_args:No \__regex_replacement_put_submatch:n
            { \int_use:N \l__regex_internal_a_int }
          \exp_after:wN \use_none:nn
        \else:
          \exp_after:wN \__regex_replacement_error:NNN
          \exp_after:wN g
        \fi:
      }
    #1 #2
  }
```

(*End of definition for* `\__regex_replacement_g:w` *and* `\__regex_replacement_g_digits:NN`.)

46.6.5 Csnames in replacement

__regex_replacement_c:w

\c may only be followed by an unescaped character. If followed by a left brace, start a control sequence by calling an auxiliary common with \u. Otherwise test whether the category is known; if it is not, complain.

```
\cs_new_protected:Npn \__regex_replacement_c:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
      {
        \cs_if_exist:cTF { __regex_replacement_c_#2:w }
          { \__regex_replacement_cat:NNN #2 }
          { \__regex_replacement_error:NNN c #1#2 }
      }
      {
        \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
          { \__regex_replacement_cu_aux:Nw \__regex_replacement_exp_not:N }
          { \__regex_replacement_error:NNN c #1#2 }
      }
  }
```

(*End of definition for* __regex_replacement_c:w.)

__regex_replacement_cu_aux:Nw

Start a control sequence with \cs:w, protected from expansion by #1 (either __regex_-replacement_exp_not:N or \exp_not:V), or turned to a string by \tl_to_str:V if inside another csname construction \c or \u. We use \tl_to_str:V rather than \tl_to_str:N to deal with integers and other registers.

```
\cs_new_protected:Npn \__regex_replacement_cu_aux:Nw #1
  {
    \if_case:w \l__regex_replacement_csnames_int
      \tl_build_put_right:Nn \l__regex_build_tl
        { \exp_not:n { \exp_after:wN #1 \cs:w } }
    \else:
      \tl_build_put_right:Nn \l__regex_build_tl
        { \exp_not:n { \exp_after:wN \tl_to_str:V \cs:w } }
    \fi:
    \int_incr:N \l__regex_replacement_csnames_int
  }
```

(*End of definition for* __regex_replacement_cu_aux:Nw.)

__regex_replacement_u:w

Check that \u is followed by a left brace. If so, start a control sequence with \cs:w, which is then unpacked either with \exp_not:V or \tl_to_str:V depending on the current context.

```
\cs_new_protected:Npn \__regex_replacement_u:w #1#2
  {
    \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
      { \__regex_replacement_cu_aux:Nw \__regex_replacement_exp_not:V }
      { \__regex_replacement_error:NNN u #1#2 }
  }
```

(*End of definition for* __regex_replacement_u:w.)

__regex_replacement_rbrace:N

Within a \c{...} or \u{...} construction, end the control sequence, and decrease the brace count. Otherwise, this is a raw right brace.

```
\cs_new_protected:Npn \__regex_replacement_rbrace:N #1
  {
    \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
      \tl_build_put_right:Nn \l__regex_build_tl { \cs_end: }
      \int_decr:N \l__regex_replacement_csnames_int
    \else:
      \__regex_replacement_normal:n {#1}
    \fi:
  }
```

(*End of definition for __regex_replacement_rbrace:N.*)

__regex_replacement_lbrace:N

Within a \c{...} or \u{...} construction, this is forbidden. Otherwise, this is a raw left brace.

```
\cs_new_protected:Npn \__regex_replacement_lbrace:N #1
  {
    \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
      \msg_error:nnn { regex } { cu-lbrace } { u }
    \else:
      \__regex_replacement_normal:n {#1}
    \fi:
  }
```

(*End of definition for __regex_replacement_lbrace:N.*)

46.6.6 Characters in replacement

__regex_replacement_cat:NNN

Here, #1 is a letter among BEMTPUDSLOA and #2#3 denote the next character. Complain if we reach the end of the replacement or if the construction appears inside \c{...} or \u{...}, and detect the case of a parenthesis. In that case, store the current category in a sequence and switch to a new one.

```
\cs_new_protected:Npn \__regex_replacement_cat:NNN #1#2#3
  {
    \token_if_eq_meaning:NNTF \prg_do_nothing: #3
      { \msg_error:nn { regex } { replacement-catcode-end } }
      {
        \int_compare:nNnTF \l__regex_replacement_csnames_int > \c_zero_int
          {
            \msg_error:nnnn
              { regex } { replacement-catcode-in-cs } {#1} {#3}
            #2 #3
          }
          {
            \__regex_two_if_eq:NNNNTF #2 #3 \__regex_replacement_normal:n (
              {
                \seq_push:NV \l__regex_replacement_category_seq
                  \l__regex_replacement_category_tl
                \tl_set:Nn \l__regex_replacement_category_tl {#1}
              }
              {
                \token_if_eq_meaning:NNT #2 \__regex_replacement_escaped:N
```

```
                    {
                      \__regex_char_if_alphanumeric:NTF #3
                        {
                          \msg_error:nnnn
                            { regex } { replacement-catcode-escaped }
                            {#1} {#3}
                        }
                        { }
                    }
                  \use:c { __regex_replacement_c_#1:w } #2 #3
                }
            }
        }
    }
```

(*End of definition for* `\__regex_replacement_cat:NNN`*.*)

We now need to change the category code of the null character many times, hence work in a group. The catcode-specific macros below are defined in alphabetical order; if you are trying to understand the code, start from the end of the alphabet as those categories are simpler than active or begin-group.

```
\group_begin:
```

`\__regex_replacement_char:nNN`

The only way to produce an arbitrary character–catcode pair is to use the `\lowercase` or `\uppercase` primitives. This is a wrapper for our purposes. The first argument is the null character with various catcodes. The second and third arguments are grabbed from the input stream: `#3` is the character whose character code to reproduce. We could use `\char_generate:nn` but only for some catcodes (active characters and spaces are not supported).

```
  \cs_new_protected:Npn \__regex_replacement_char:nNN #1#2#3
    {
      \tex_lccode:D \c_zero_int = `#3 \scan_stop:
      \tex_lowercase:D { \__regex_replacement_put:n {#1} }
    }
```

(*End of definition for* `\__regex_replacement_char:nNN`*.*)

`\__regex_replacement_c_A:w`

For an active character, expansion must be avoided, twice because we later do two `e`-expansions, to unpack `\toks` for the query, and to expand their contents to tokens of the query.

```
  \char_set_catcode_active:N \^^@
  \cs_new_protected:Npn \__regex_replacement_c_A:w
    { \__regex_replacement_char:nNN { \exp_not:n { \exp_not:N ^^@ } } }
```

(*End of definition for* `\__regex_replacement_c_A:w`*.*)

`\__regex_replacement_c_B:w`

An explicit begin-group token increases the balance, unless within a `\c{...}` or `\u{...}` construction. Add the desired begin-group character, using the standard `\if_false:` trick. We eventually `e`-expand twice. The first time must yield a balanced token list, and the second one gives the bare begin-group token. The `\exp_after:wN` is not strictly needed, but is more consistent with l3tl-analysis.

```
  \char_set_catcode_group_begin:N \^^@
  \cs_new_protected:Npn \__regex_replacement_c_B:w
    {
```

```
      \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
        \int_incr:N \l__regex_balance_int
      \fi:
      \__regex_replacement_char:nNN
        { \exp_not:n { \exp_after:wN ^^@ \if_false: } \fi: } }
    }
```

(*End of definition for* `\__regex_replacement_c_B:w`.)

`\__regex_replacement_c_C:w`

This is not quite catcode-related: when the user requests a character with category "control sequence", the one-character control symbol is returned. As for the active character, we prepare for two `e`-expansions.

```
  \cs_new_protected:Npn \__regex_replacement_c_C:w #1#2
    {
      \tl_build_put_right:Nn \l__regex_build_tl
        { \exp_not:N \__regex_replacement_exp_not:N \exp_not:c {#2} }
    }
```

(*End of definition for* `\__regex_replacement_c_C:w`.)

`\__regex_replacement_c_D:w`

Subscripts fit the mould: `\lowercase` the null byte with the correct category.

```
  \char_set_catcode_math_subscript:N \^^@
  \cs_new_protected:Npn \__regex_replacement_c_D:w
    { \__regex_replacement_char:nNN { ^^@ } }
```

(*End of definition for* `\__regex_replacement_c_D:w`.)

`\__regex_replacement_c_E:w`

Similar to the begin-group case, the second `e`-expansion produces the bare end-group token.

```
  \char_set_catcode_group_end:N \^^@
  \cs_new_protected:Npn \__regex_replacement_c_E:w
    {
      \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
        \int_decr:N \l__regex_balance_int
      \fi:
      \__regex_replacement_char:nNN
        { \exp_not:n { \if_false: { \fi:  ^^@ } }
    }
```

(*End of definition for* `\__regex_replacement_c_E:w`.)

`\__regex_replacement_c_L:w`

Simply `\lowercase` a letter null byte to produce an arbitrary letter.

```
  \char_set_catcode_letter:N \^^@
  \cs_new_protected:Npn \__regex_replacement_c_L:w
    { \__regex_replacement_char:nNN { ^^@ } }
```

(*End of definition for* `\__regex_replacement_c_L:w`.)

`\__regex_replacement_c_M:w`

No surprise here, we lowercase the null math toggle.

```
  \char_set_catcode_math_toggle:N \^^@
  \cs_new_protected:Npn \__regex_replacement_c_M:w
    { \__regex_replacement_char:nNN { ^^@ } }
```

(*End of definition for* `\__regex_replacement_c_M:w`.)

__regex_replacement_c_O:w Lowercase an other null byte.

```
\char_set_catcode_other:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_O:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End of definition for* `\__regex_replacement_c_O:w`.)

__regex_replacement_c_P:w For macro parameters, expansion is a tricky issue. We need to prepare for two `e`-expansions and passing through various macro definitions. Note that we cannot replace one `\exp_not:n` by doubling the macro parameter characters because this would misbehave if a mischievous user asks for `\c{\cP\#}`, since that macro parameter character would be doubled.

```
\char_set_catcode_parameter:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_P:w
  {
    \__regex_replacement_char:nNN
      { \exp_not:n { \exp_not:n { ^^@^^@^^@^^@ } } }
  }
```

(*End of definition for* `\__regex_replacement_c_P:w`.)

__regex_replacement_c_S:w Spaces are normalized on input by TeX to have character code 32. It is in fact impossible to get a token with character code 0 and category code 10. Hence we use 32 instead of 0 as our base character.

```
\cs_new_protected:Npn \__regex_replacement_c_S:w #1#2
  {
    \if_int_compare:w `#2 = \c_zero_int
      \msg_error:nn { regex } { replacement-null-space }
    \fi:
    \tex_lccode:D `\ = `#2 \scan_stop:
    \tex_lowercase:D { \__regex_replacement_put:n {~} }
  }
```

(*End of definition for* `\__regex_replacement_c_S:w`.)

__regex_replacement_c_T:w No surprise for alignment tabs here. Those are surrounded by the appropriate braces whenever necessary, hence they don't cause trouble in alignment settings.

```
\char_set_catcode_alignment:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_T:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End of definition for* `\__regex_replacement_c_T:w`.)

__regex_replacement_c_U:w Simple call to `\__regex_replacement_char:nNN` which lowercases the math superscript `^^@`.

```
\char_set_catcode_math_superscript:N \^^@
\cs_new_protected:Npn \__regex_replacement_c_U:w
  { \__regex_replacement_char:nNN { ^^@ } }
```

(*End of definition for* `\__regex_replacement_c_U:w`.)

Restore the catcode of the null byte.

```
\group_end:
```

46.6.7 An error

__regex_replacement_error:NNN

Simple error reporting by calling one of the messages `replacement-c`, `replacement-g`, or `replacement-u`.

```
\cs_new_protected:Npn \__regex_replacement_error:NNN #1#2#3
  {
    \msg_error:nne { regex } { replacement-#1 } {#3}
    #2 #3
  }
```

(*End of definition for* `\__regex_replacement_error:NNN`*.*)

46.7 User functions

\regex_new:N

Before being assigned a sensible value, a regex variable matches nothing.

```
\cs_new_protected:Npn \regex_new:N #1
  { \cs_new_eq:NN #1 \c__regex_no_match_regex }
```

(*End of definition for* `\regex_new:N`*. This function is documented on page 56.*)

\l_tmpa_regex
\l_tmpb_regex
\g_tmpa_regex
\g_tmpb_regex

The usual scratch space.

```
\regex_new:N \l_tmpa_regex
\regex_new:N \l_tmpb_regex
\regex_new:N \g_tmpa_regex
\regex_new:N \g_tmpb_regex
```

(*End of definition for* `\l_tmpa_regex` *and others. These variables are documented on page 61.*)

\regex_set:Nn
\regex_gset:Nn
\regex_const:Nn

Compile, then store the result in the user variable with the appropriate assignment function.

```
\cs_new_protected:Npn \regex_set:Nn #1#2
  {
    \__regex_compile:n {#2}
    \tl_set_eq:NN #1 \l__regex_internal_regex
  }
\cs_new_protected:Npn \regex_gset:Nn #1#2
  {
    \__regex_compile:n {#2}
    \tl_gset_eq:NN #1 \l__regex_internal_regex
  }
\cs_new_protected:Npn \regex_const:Nn #1#2
  {
    \__regex_compile:n {#2}
    \tl_const:Ne #1 { \exp_not:o \l__regex_internal_regex }
  }
```

(*End of definition for* `\regex_set:Nn`*,* `\regex_gset:Nn`*, and* `\regex_const:Nn`*. These functions are documented on page 56.*)

\regex_show:n
\regex_log:n
__regex_show:Nn
\regex_show:N
\regex_log:N
__regex_show:NN

User functions: the `n` variant requires compilation first. Then show the variable with some appropriate text. The auxiliary `\__regex_show:N` is defined in a different section.

```
\cs_new_protected:Npn \regex_show:n { \__regex_show:Nn \msg_show:nneeee }
\cs_new_protected:Npn \regex_log:n { \__regex_show:Nn \msg_log:nneeee }
\cs_new_protected:Npn \__regex_show:Nn #1#2
```

```
  {
    \__regex_compile:n {#2}
    \__regex_show:N \l__regex_internal_regex
    #1 { regex } { show }
      { \tl_to_str:n {#2} } { }
      { \l__regex_internal_a_tl } { }
  }
\cs_new_protected:Npn \regex_show:N { \__regex_show:NN \msg_show:nneeee }
\cs_new_protected:Npn \regex_log:N { \__regex_show:NN \msg_log:nneeee }
\cs_new_protected:Npn \__regex_show:NN #1#2
  {
    \__kernel_chk_tl_type:NnnT #2 { regex }
      { \exp_args:No \__regex_clean_regex:n {#2} }
      {
        \__regex_show:N #2
        #1 { regex } { show }
          { } { \token_to_str:N #2 }
          { \l__regex_internal_a_tl } { }
      }
  }
```

(*End of definition for* `\regex_show:n` *and others. These functions are documented on page 56.*)

`\regex_match:nnTF`
`\regex_match:nVTF`
`\regex_match:NnTF`
`\regex_match:NVTF`

Those conditionals are based on a common auxiliary defined later. Its first argument builds the NFA corresponding to the regex, and the second argument is the query token list. Once we have performed the match, convert the resulting boolean to `\prg_return_-true:` or `false`.

```
\prg_new_protected_conditional:Npnn \regex_match:nn #1#2 { T , F , TF }
  {
    \__regex_if_match:nn { \__regex_build:n {#1} } {#2}
    \__regex_return:
  }
\prg_generate_conditional_variant:Nnn \regex_match:nn { nV } { T , F , TF }
\prg_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
  {
    \__regex_if_match:nn { \__regex_build:N #1 } {#2}
    \__regex_return:
  }
\prg_generate_conditional_variant:Nnn \regex_match:Nn { NV } { T , F , TF }
```

(*End of definition for* `\regex_match:nnTF` *and* `\regex_match:NnTF`*. These functions are documented on page 57.*)

`\regex_count:nnN`
`\regex_count:nVN`
`\regex_count:NnN`
`\regex_count:NVN`

Again, use an auxiliary whose first argument builds the NFA.

```
\cs_new_protected:Npn \regex_count:nnN #1
  { \__regex_count:nnN { \__regex_build:n {#1} } }
\cs_new_protected:Npn \regex_count:NnN #1
  { \__regex_count:nnN { \__regex_build:N #1 } }
\cs_generate_variant:Nn \regex_count:nnN { nV }
\cs_generate_variant:Nn \regex_count:NnN { NV }
```

(*End of definition for* `\regex_count:nnN` *and* `\regex_count:NnN`*. These functions are documented on page 57.*)

\regex_match_case:nn
\regex_match_case:nnTF

The auxiliary errors if #1 has an odd number of items, and otherwise it sets \g__regex_case_int according to which case was found (zero if not found). The true branch leaves the corresponding code in the input stream.

```
\cs_new_protected:Npn \regex_match_case:nnTF #1#2#3
  {
    \__regex_match_case:nnTF {#1} {#2}
      {
        \tl_item:nn {#1} { 2 * \g__regex_case_int }
        #3
      }
  }
\cs_new_protected:Npn \regex_match_case:nn #1#2
  { \regex_match_case:nnTF {#1} {#2} { } { } }
\cs_new_protected:Npn \regex_match_case:nnT #1#2#3
  { \regex_match_case:nnTF {#1} {#2} {#3} { } }
\cs_new_protected:Npn \regex_match_case:nnF #1#2
  { \regex_match_case:nnTF {#1} {#2} { } }
```

(*End of definition for* \regex_match_case:nnTF*. This function is documented on page 57.*)

\regex_extract_once:nnN
\regex_extract_once:nVN
\regex_extract_once:nnNTF
\regex_extract_once:nVNTF
\regex_extract_once:NnN
\regex_extract_once:NVN
\regex_extract_once:NnNTF
\regex_extract_once:NVNTF
\regex_extract_all:nnN
\regex_extract_all:nVN
\regex_extract_all:nnNTF
\regex_extract_all:nVNTF
\regex_extract_all:NnN
\regex_extract_all:NVN
\regex_extract_all:NnNTF
\regex_extract_all:NVNTF
\regex_replace_once:nnN
\regex_replace_once:nVN
\regex_replace_once:nnNTF
\regex_replace_once:nVNTF
\regex_replace_once:NnN
\regex_replace_once:NVN
\regex_replace_once:NnNTF
\regex_replace_once:NVNTF
\regex_replace_all:nnN
\regex_replace_all:nVN
\regex_replace_all:nnNTF
\regex_replace_all:nVNTF
\regex_replace_all:NnN
\regex_replace_all:NVN
\regex_replace_all:NnNTF
\regex_replace_all:NVNTF
\regex_split:NnN
\regex_split:NVN
\regex_split:NnNTF
\regex_split:NVNTF
\regex_split:nnN
\regex_split:nVN
\regex_split:nnNTF
\regex_split:nVNTF

We define here 40 user functions, following a common pattern in terms of :nnN auxiliaries, defined in the coming subsections. The auxiliary is handed __regex_build:n or __regex_build:N with the appropriate regex argument, then all other necessary arguments (replacement text, token list, *etc.* The conditionals call __regex_return: to return either true or false once matching has been performed.

```
\cs_set_protected:Npn \__regex_tmp:w #1#2#3
  {
    \cs_new_protected:Npn #2 ##1 { #1 { \__regex_build:n {##1} } }
    \cs_new_protected:Npn #3 ##1 { #1 { \__regex_build:N  ##1  } }
    \prg_new_protected_conditional:Npnn #2 ##1##2##3 { T , F , TF }
      { #1 { \__regex_build:n {##1} } {##2} ##3 \__regex_return: }
    \prg_new_protected_conditional:Npnn #3 ##1##2##3 { T , F , TF }
      { #1 { \__regex_build:N  ##1  } {##2} ##3 \__regex_return: }
    \cs_generate_variant:Nn #2 { nV }
    \prg_generate_conditional_variant:Nnn #2 { nV } { T , F , TF }
    \cs_generate_variant:Nn #3 { NV }
    \prg_generate_conditional_variant:Nnn #3 { NV } { T , F , TF }
  }
\__regex_tmp:w \__regex_extract_once:nnN
  \regex_extract_once:nnN \regex_extract_once:NnN
\__regex_tmp:w \__regex_extract_all:nnN
  \regex_extract_all:nnN \regex_extract_all:NnN
\__regex_tmp:w \__regex_replace_once:nnN
  \regex_replace_once:nnN \regex_replace_once:NnN
\__regex_tmp:w \__regex_replace_all:nnN
  \regex_replace_all:nnN \regex_replace_all:NnN
\__regex_tmp:w \__regex_split:nnN \regex_split:nnN \regex_split:NnN
```

(*End of definition for* \regex_extract_once:nnNTF *and others. These functions are documented on page 58.*)

\regex_replace_case_once:nN
\regex_replace_case_once:nNTF

If the input is bad (odd number of items) then take the false branch. Otherwise, use the same auxiliary as \regex_replace_once:nnN, but with more complicated code to build the automaton, and to find what replacement text to use. The `\tl_item:nn` is only expanded once we know the value of `\g__regex_case_int`, namely which case matched.

```
\cs_new_protected:Npn \regex_replace_case_once:nNTF #1#2
  {
    \int_if_odd:nTF { \tl_count:n {#1} }
      {
        \msg_error:nneeee { regex } { case-odd }
          { \token_to_str:N \regex_replace_case_once:nN(TF) } { code }
          { \tl_count:n {#1} } { \tl_to_str:n {#1} }
        \use_ii:nn
      }
      {
        \__regex_replace_once_aux:nnN
          { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
          { \__regex_replacement:e { \tl_item:nn {#1} { 2 * \g__regex_case_int } } }
          #2
        \bool_if:NTF \g__regex_success_bool
      }
  }
\cs_new_protected:Npn \regex_replace_case_once:nN #1#2
  { \regex_replace_case_once:nNTF {#1} {#2} { } { } }
\cs_new_protected:Npn \regex_replace_case_once:nNT #1#2#3
  { \regex_replace_case_once:nNTF {#1} {#2} {#3} { } }
\cs_new_protected:Npn \regex_replace_case_once:nNF #1#2
  { \regex_replace_case_once:nNTF {#1} {#2} { } }
```

(*End of definition for* `\regex_replace_case_once:nNTF`*. This function is documented on page 60.*)

`\regex_replace_case_all:nN`
`\regex_replace_case_all:nNTF`

If the input is bad (odd number of items) then take the false branch. Otherwise, use the same auxiliary as `\regex_replace_all:nnN`, but with more complicated code to build the automaton, and to find what replacement text to use.

```
\cs_new_protected:Npn \regex_replace_case_all:nNTF #1#2
  {
    \int_if_odd:nTF { \tl_count:n {#1} }
      {
        \msg_error:nneeee { regex } { case-odd }
          { \token_to_str:N \regex_replace_case_all:nN(TF) } { code }
          { \tl_count:n {#1} } { \tl_to_str:n {#1} }
        \use_ii:nn
      }
      {
        \__regex_replace_all_aux:nnN
          { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
          { \__regex_case_replacement:e { \__regex_tl_even_items:n {#1} } }
          #2
        \bool_if:NTF \g__regex_success_bool
      }
  }
\cs_new_protected:Npn \regex_replace_case_all:nN #1#2
  { \regex_replace_case_all:nNTF {#1} {#2} { } { } }
\cs_new_protected:Npn \regex_replace_case_all:nNT #1#2#3
  { \regex_replace_case_all:nNTF {#1} {#2} {#3} { } }
\cs_new_protected:Npn \regex_replace_case_all:nNF #1#2
  { \regex_replace_case_all:nNTF {#1} {#2} { } }
```

(*End of definition for* `\regex_replace_case_all:nNTF`. *This function is documented on page* 60.)

46.7.1 Variables and helpers for user functions

`\l__regex_match_count_int`

The number of matches found so far is stored in `\l__regex_match_count_int`. This is only used in the `\regex_count:nnN` functions.

```
\int_new:N \l__regex_match_count_int
```

(*End of definition for* `\l__regex_match_count_int`.)

`\l__regex_begin_flag`
`\l__regex_end_flag`

Those flags are raised to indicate begin-group or end-group tokens that had to be added when extracting submatches.

```
\flag_new:N \l__regex_begin_flag
\flag_new:N \l__regex_end_flag
```

(*End of definition for* `\l__regex_begin_flag` *and* `\l__regex_end_flag`.)

`\l__regex_min_submatch_int`
`\l__regex_submatch_int`
`\l__regex_zeroth_submatch_int`

The end-points of each submatch are stored in two arrays whose index ⟨*submatch*⟩ ranges from `\l__regex_min_submatch_int` (inclusive) to `\l__regex_submatch_int` (exclusive). Each successful match comes with a 0-th submatch (the full match), and one match for each capturing group: submatches corresponding to the last successful match are labelled starting at `zeroth_submatch`. The entry `\l__regex_zeroth_submatch_int` in `\g__regex_submatch_prev_intarray` holds the position at which that match attempt started: this is used for splitting and replacements.

```
\int_new:N \l__regex_min_submatch_int
\int_new:N \l__regex_submatch_int
\int_new:N \l__regex_zeroth_submatch_int
```

(*End of definition for* `\l__regex_min_submatch_int`, `\l__regex_submatch_int`, *and* `\l__regex_zeroth_submatch_int`.)

`\g__regex_submatch_prev_intarray`
`\g__regex_submatch_begin_intarray`
`\g__regex_submatch_end_intarray`
`\g__regex_submatch_case_intarray`

Hold the place where the match attempt begun, the end-points of each submatch, and which regex case the match corresponds to, respectively.

```
\intarray_new:Nn \g__regex_submatch_prev_intarray { 65536 }
\intarray_new:Nn \g__regex_submatch_begin_intarray { 65536 }
\intarray_new:Nn \g__regex_submatch_end_intarray { 65536 }
\intarray_new:Nn \g__regex_submatch_case_intarray { 65536 }
```

(*End of definition for* `\g__regex_submatch_prev_intarray` *and others.*)

`\g__regex_balance_intarray`

The first thing we do when matching is to store the balance of begin-group/end-group characters into `\g__regex_balance_intarray`.

```
\intarray_new:Nn \g__regex_balance_intarray { 65536 }
```

(*End of definition for* `\g__regex_balance_intarray`.)

`\l__regex_added_begin_int`
`\l__regex_added_end_int`

Keep track of the number of left/right braces to add when performing a regex operation such as a replacement.

```
\int_new:N \l__regex_added_begin_int
\int_new:N \l__regex_added_end_int
```

(*End of definition for* `\l__regex_added_begin_int` *and* `\l__regex_added_end_int`.)

__regex_return: This function triggers either \prg_return_false: or \prg_return_true: as appropriate to whether a match was found or not. It is used by all user conditionals.

```
\cs_new_protected:Npn \__regex_return:
  {
    \if_meaning:w \c_true_bool \g__regex_success_bool
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* __regex_return:.)

__regex_query_set:n __regex_query_set_aux:nN To easily extract subsets of the input once we found the positions at which to cut, store the input tokens one by one into successive \toks registers. Also store the brace balance (used to check for overall brace balance) in an array.

```
\cs_new_protected:Npn \__regex_query_set:n #1
  {
    \int_zero:N \l__regex_balance_int
    \int_zero:N \l__regex_curr_pos_int
    \__regex_query_set_aux:nN { } F
    \tl_analysis_map_inline:nn {#1}
      { \__regex_query_set_aux:nN {##1} ##3 }
    \__regex_query_set_aux:nN { } F
    \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
  }
\cs_new_protected:Npn \__regex_query_set_aux:nN #1#2
  {
    \int_incr:N \l__regex_curr_pos_int
    \__regex_toks_set:Nn \l__regex_curr_pos_int {#1}
    \__kernel_intarray_gset:Nnn \g__regex_balance_intarray
      \l__regex_curr_pos_int \l__regex_balance_int
    \if_case:w "#2 \exp_stop_f:
    \or: \int_incr:N \l__regex_balance_int
    \or: \int_decr:N \l__regex_balance_int
    \fi:
  }
```

(*End of definition for* __regex_query_set:n *and* __regex_query_set_aux:nN.)

46.7.2 Matching

__regex_if_match:nn We don't track submatches, and stop after a single match. Build the NFA with #1, and perform the match on the query #2.

```
\cs_new_protected:Npn \__regex_if_match:nn #1#2
  {
    \group_begin:
      \__regex_disable_submatches:
      \__regex_single_match:
      #1
      \__regex_match:n {#2}
    \group_end:
  }
```

(*End of definition for* `\__regex_if_match:nn`.)

`\__regex_match_case:nnTF`
`\__regex_match_case_aux:nn`

The code would get badly messed up if the number of items in `#1` were not even, so we catch this case, then follow the same code as `\regex_match:nnTF` but using `\__regex_case_build:n` and without returning a result.

```
\cs_new_protected:Npn \__regex_match_case:nnTF #1#2
  {
    \int_if_odd:nTF { \tl_count:n {#1} }
      {
        \msg_error:nneeee { regex } { case-odd }
          { \token_to_str:N \regex_match_case:nn(TF) } { code }
          { \tl_count:n {#1} } { \tl_to_str:n {#1} }
        \use_ii:nn
      }
      {
        \__regex_if_match:nn
          { \__regex_case_build:e { \__regex_tl_odd_items:n {#1} } }
          {#2}
        \bool_if:NTF \g__regex_success_bool
      }
  }
\cs_new:Npn \__regex_match_case_aux:nn #1#2 { \exp_not:n { {#1} } }
```

(*End of definition for* `\__regex_match_case:nnTF` *and* `\__regex_match_case_aux:nn`.)

`\__regex_count:nnN`

Again, we don't care about submatches. Instead of aborting after the first "longest match" is found, we search for multiple matches, incrementing `\l__regex_match_count_int` every time to record the number of matches. Build the NFA and match. At the end, store the result in the user's variable.

```
\cs_new_protected:Npn \__regex_count:nnN #1#2#3
  {
    \group_begin:
      \__regex_disable_submatches:
      \int_zero:N \l__regex_match_count_int
      \__regex_multi_match:n { \int_incr:N \l__regex_match_count_int }
      #1
      \__regex_match:n {#2}
      \exp_args:NNNo
    \group_end:
    \int_set:Nn #3 { \int_use:N \l__regex_match_count_int }
  }
```

(*End of definition for* `\__regex_count:nnN`.)

46.7.3 Extracting submatches

`\__regex_extract_once:nnN`
`\__regex_extract_all:nnN`

Match once or multiple times. After each match (or after the only match), extract the submatches using `\__regex_extract:`. At the end, store the sequence containing all the submatches into the user variable `#3` after closing the group.

```
\cs_new_protected:Npn \__regex_extract_once:nnN #1#2#3
  {
    \group_begin:
      \__regex_single_match:
```

```
      #1
      \__regex_match:n {#2}
      \__regex_extract:
      \__regex_query_set:n {#2}
    \__regex_group_end_extract_seq:N #3
  }
\cs_new_protected:Npn \__regex_extract_all:nnN #1#2#3
  {
    \group_begin:
      \__regex_multi_match:n { \__regex_extract: }
      #1
      \__regex_match:n {#2}
      \__regex_query_set:n {#2}
    \__regex_group_end_extract_seq:N #3
  }
```

(*End of definition for* `\__regex_extract_once:nnN` *and* `\__regex_extract_all:nnN`.)

`\__regex_split:nnN` Splitting at submatches is a bit more tricky. For each match, extract all submatches, and replace the zeroth submatch by the part of the query between the start of the match attempt and the start of the zeroth submatch. This is inhibited if the delimiter matched an empty token list at the start of this match attempt. After the last match, store the last part of the token list, which ranges from the start of the match attempt to the end of the query. This step is inhibited if the last match was empty and at the very end: decrement `\l__regex_submatch_int`, which controls which matches will be used.

```
\cs_new_protected:Npn \__regex_split:nnN #1#2#3
  {
    \group_begin:
      \__regex_multi_match:n
        {
          \if_int_compare:w
            \l__regex_start_pos_int < \l__regex_success_pos_int
            \__regex_extract:
            \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
              \l__regex_zeroth_submatch_int \c_zero_int
            \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
              \l__regex_zeroth_submatch_int
              {
                \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray
                  \l__regex_zeroth_submatch_int
              }
            \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
              \l__regex_zeroth_submatch_int
              \l__regex_start_pos_int
          \fi:
        }
      #1
      \__regex_match:n {#2}
      \__regex_query_set:n {#2}
      \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
        \l__regex_submatch_int \c_zero_int
      \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
        \l__regex_submatch_int
        \l__regex_max_pos_int
```

```
      \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
        \l__regex_submatch_int
        \l__regex_start_pos_int
      \int_incr:N \l__regex_submatch_int
      \if_meaning:w \c_true_bool \l__regex_empty_success_bool
        \if_int_compare:w \l__regex_start_pos_int = \l__regex_max_pos_int
          \int_decr:N \l__regex_submatch_int
        \fi:
      \fi:
    \__regex_group_end_extract_seq:N #3
  }
```

(*End of definition for* `\__regex_split:nnN`.)

`\__regex_group_end_extract_seq:N`
`\__regex_extract_seq:N`
`\__regex_extract_seq:NNn`
`\__regex_extract_seq_loop:Nw`

The end-points of submatches are stored as entries of two arrays from `\l__regex_min_submatch_int` to `\l__regex_submatch_int` (exclusive). Extract the relevant ranges into `\g__regex_internal_tl`, separated by `\__regex_tmp:w {}`. We keep track in the two flags `__regex_begin` and `__regex_end` of the number of begin-group or end-group tokens added to make each of these items overall balanced. At this step, `}{` is counted as being balanced (same number of begin-group and end-group tokens). This problem is caught by `\__regex_extract_check:w`, explained later. After complaining about any begin-group or end-group tokens we had to add, we are ready to construct the user's sequence outside the group.

```
\cs_new_protected:Npn \__regex_group_end_extract_seq:N #1
  {
      \flag_clear:N \l__regex_begin_flag
      \flag_clear:N \l__regex_end_flag
      \cs_set_eq:NN \__regex_tmp:w \scan_stop:
      \__kernel_tl_gset:Nx \g__regex_internal_tl
        {
          \int_step_function:nnN \l__regex_min_submatch_int
            { \l__regex_submatch_int - \c_one_int } \__regex_extract_seq_aux:n
          \__regex_tmp:w
        }
      \int_set:Nn \l__regex_added_begin_int
        { \flag_height:N \l__regex_begin_flag }
      \int_set:Nn \l__regex_added_end_int
        { \flag_height:N \l__regex_end_flag }
      \tex_afterassignment:D \__regex_extract_check:w
      \__kernel_tl_gset:Nx \g__regex_internal_tl
        { \g__regex_internal_tl \if_false: { \fi: } }
      \int_compare:nNnT
        { \l__regex_added_begin_int + \l__regex_added_end_int } > \c_zero_int
        {
          \msg_error:nneee { regex } { result-unbalanced }
            { splitting~or~extracting~submatches }
            { \int_use:N \l__regex_added_begin_int }
            { \int_use:N \l__regex_added_end_int }
        }
    \group_end:
    \__regex_extract_seq:N #1
  }
\cs_gset_protected:Npn \__regex_extract_seq:N #1
  {
```

```
    \seq_clear:N #1
    \cs_set_eq:NN \__regex_tmp:w  \__regex_extract_seq_loop:Nw
    \exp_after:wN \__regex_extract_seq:NNn
    \exp_after:wN #1
    \g__regex_internal_tl \use_none:nnn
  }
\cs_new_protected:Npn \__regex_extract_seq:NNn #1#2#3
  { #3 #2 #1 \prg_do_nothing: }
\cs_new_protected:Npn \__regex_extract_seq_loop:Nw #1#2 \__regex_tmp:w #3
  {
    \seq_put_right:No #1 {#2}
    #3 \__regex_extract_seq_loop:Nw #1 \prg_do_nothing:
  }
```

(*End of definition for* `\__regex_group_end_extract_seq:N` *and others.*)

`\__regex_extract_seq_aux:n`
`\__regex_extract_seq_aux:ww`

The `:n` auxiliary builds one item of the sequence of submatches. First compute the brace balance of the submatch, then extract the submatch from the query, adding the appropriate braces and raising a flag if the submatch is not balanced.

```
\cs_new:Npn \__regex_extract_seq_aux:n #1
  {
    \__regex_tmp:w { }
    \exp_after:wN \__regex_extract_seq_aux:ww
    \int_value:w \__regex_submatch_balance:n {#1} ; #1;
  }
\cs_new:Npn \__regex_extract_seq_aux:ww #1; #2;
  {
    \if_int_compare:w #1 < \c_zero_int
      \prg_replicate:nn {-#1}
        {
          \flag_raise:N \l__regex_begin_flag
          \exp_not:n { { \if_false: } \fi: }
        }
    \fi:
    \__regex_query_submatch:n {#2}
    \if_int_compare:w #1 > \c_zero_int
      \prg_replicate:nn {#1}
        {
          \flag_raise:N \l__regex_end_flag
          \exp_not:n { \if_false: { \fi: } }
        }
    \fi:
  }
```

(*End of definition for* `\__regex_extract_seq_aux:n` *and* `\__regex_extract_seq_aux:ww`.)

`\__regex_extract_check:w`
`\__regex_extract_check:n`
`\__regex_extract_check_loop:w`
`\__regex_extract_check_end:w`

In `\__regex_group_end_extract_seq:N` we had to expand `\g__regex_internal_tl` to turn `\if_false:` constructions into actual begin-group and end-group tokens. This is done with a `\__kernel_tl_gset:Nx` assignment, and `\__regex_extract_check:w` is run immediately after this assignment ends, thanks to the `\afterassignment` primitive. If all of the items were properly balanced (enough begin-group tokens before end-group tokens, so `}{` is not) then `\__regex_extract_check:w` is called just before the closing brace of the `\__kernel_tl_gset:Nx` (thanks to our sneaky `\if_false: { \fi: }` construction), and finds that there is nothing left to expand. If any of the items is unbalanced, the assignment gets ended early by an extra end-group token, and our check finds more tokens needing to be expanded in a new `\__kernel_tl_gset:Nx` assignment. We need to add a begin-group and an end-group tokens to the unbalanced item, namely to the last item found so far, which we reach through a loop.

```
\cs_new_protected:Npn \__regex_extract_check:w
  {
    \exp_after:wN \__regex_extract_check:n
    \exp_after:wN { \if_false: } \fi:
  }
\cs_new_protected:Npn \__regex_extract_check:n #1
  {
    \tl_if_empty:nF {#1}
      {
        \int_incr:N \l__regex_added_begin_int
        \int_incr:N \l__regex_added_end_int
        \tex_afterassignment:D \__regex_extract_check:w
        \__kernel_tl_gset:Nx \g__regex_internal_tl
          {
            \exp_after:wN \__regex_extract_check_loop:w
            \g__regex_internal_tl
            \__regex_tmp:w \__regex_extract_check_end:w
            #1
          }
      }
  }
\cs_new:Npn \__regex_extract_check_loop:w #1 \__regex_tmp:w #2
  {
    #2
    \exp_not:o {#1}
    \__regex_tmp:w { }
    \__regex_extract_check_loop:w \prg_do_nothing:
  }
```

Arguments of `\__regex_extract_check_end:w` are: `#1` is the part of the item before the extra end-group token; `#2` is junk; `#3` is `\prg_do_nothing:` followed by the not-yet-expanded part of the item after the extra end-group token. In the replacement text, the first brace and the `\if_false: { \fi: }` construction are the added begin-group and end-group tokens (the latter being not-yet expanded, just like `#3`), while the closing brace after `\exp_not:o {#1}` replaces the extra end-group token that had ended the assignment early. In particular this means that the character code of that end-group token is lost.

```
\cs_new:Npn \__regex_extract_check_end:w
    \exp_not:o #1#2 \__regex_extract_check_loop:w #3 \__regex_tmp:w
  {
    { \exp_not:o {#1} }
    #3
    \if_false: { \fi: }
    \__regex_tmp:w
  }
```

(*End of definition for* `\__regex_extract_check:w` *and others.*)

`\__regex_extract:`
`\__regex_extract_aux:w`

Our task here is to store the list of end-points of submatches, and store them in appropriate array entries, from `\l__regex_zeroth_submatch_int` upwards. First, we store in `\g__regex_submatch_prev_intarray` the position at which the match attempt started.

We extract the rest from the comma list \l__regex_success_submatches_tl, which starts with entries to be stored in \g__regex_submatch_begin_intarray and continues with entries for \g__regex_submatch_end_intarray.

```
\cs_new_protected:Npn \__regex_extract:
  {
    \if_meaning:w \c_true_bool \g__regex_success_bool
      \int_set_eq:NN \l__regex_zeroth_submatch_int \l__regex_submatch_int
      \prg_replicate:nn \l__regex_capturing_group_int
        {
          \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
            \l__regex_submatch_int \c_zero_int
          \__kernel_intarray_gset:Nnn \g__regex_submatch_case_intarray
            \l__regex_submatch_int \c_zero_int
          \int_incr:N \l__regex_submatch_int
        }
      \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
        \l__regex_zeroth_submatch_int \l__regex_start_pos_int
      \__kernel_intarray_gset:Nnn \g__regex_submatch_case_intarray
        \l__regex_zeroth_submatch_int \g__regex_case_int
      \int_zero:N \l__regex_internal_a_int
      \exp_after:wN \__regex_extract_aux:w \l__regex_success_submatches_tl
        \prg_break_point: \__regex_use_none_delimit_by_q_recursion_stop:w ,
        \q__regex_recursion_stop
    \fi:
  }
\cs_new_protected:Npn \__regex_extract_aux:w #1 ,
  {
    \prg_break: #1 \prg_break_point:
    \if_int_compare:w \l__regex_internal_a_int < \l__regex_capturing_group_int
      \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
        { \__regex_int_eval:w \l__regex_zeroth_submatch_int + \l__regex_internal_a_int } {#
    \else:
      \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
        {
          \__regex_int_eval:w
            \l__regex_zeroth_submatch_int + \l__regex_internal_a_int
            - \l__regex_capturing_group_int
        }
        {#1}
    \fi:
    \int_incr:N \l__regex_internal_a_int
    \__regex_extract_aux:w
  }
```

(*End of definition for* __regex_extract: *and* __regex_extract_aux:w.)

46.7.4 Replacement

__regex_replace_once:nnN
__regex_replace_once_aux:nnN

Build the NFA and the replacement functions, then find a single match. If the match failed, simply exit the group. Otherwise, we do the replacement. Extract submatches. Compute the brace balance corresponding to replacing this match by the replacement (this depends on submatches). Prepare the replaced token list: the replacement function produces the tokens from the start of the query to the start of the match and the replacement text for this match; we need to add the tokens from the end of the match to the end of the query. Finally, store the result in the user's variable after closing the group: this step involves an additional e-expansion, and checks that braces are balanced in the final result.

```
\cs_new_protected:Npn \__regex_replace_once:nnN #1#2
  { \__regex_replace_once_aux:nnN {#1} { \__regex_replacement:n {#2} } }
\cs_new_protected:Npn \__regex_replace_once_aux:nnN #1#2#3
  {
    \group_begin:
      \__regex_single_match:
      #1
      \exp_args:No \__regex_match:n {#3}
    \bool_if:NTF \g__regex_success_bool
      {
        \__regex_extract:
        \exp_args:No \__regex_query_set:n {#3}
        #2
        \int_set:Nn \l__regex_balance_int
          { \__regex_replacement_balance_one_match:n \l__regex_zeroth_submatch_int }
        \__kernel_tl_set:Nx \l__regex_internal_a_tl
          {
            \__regex_replacement_do_one_match:n \l__regex_zeroth_submatch_int
            \__regex_query_range:nn
              {
                \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
                  \l__regex_zeroth_submatch_int
              }
              \l__regex_max_pos_int
          }
        \__regex_group_end_replace:N #3
      }
      { \group_end: }
  }
```

(*End of definition for* `\__regex_replace_once:nnN` *and* `\__regex_replace_once_aux:nnN`*.*)

`\__regex_replace_all:nnN` Match multiple times, and for every match, extract submatches and additionally store the position at which the match attempt started. The entries from `\l__regex_min_submatch_int` to `\l__regex_submatch_int` hold information about submatches of every match in order; each match corresponds to `\l__regex_capturing_group_int` consecutive entries. Compute the brace balance corresponding to doing all the replacements: this is the sum of brace balances for replacing each match. Join together the replacement texts for each match (including the part of the query before the match), and the end of the query.

```
\cs_new_protected:Npn \__regex_replace_all:nnN #1#2
  { \__regex_replace_all_aux:nnN {#1} { \__regex_replacement:n {#2} } }
\cs_new_protected:Npn \__regex_replace_all_aux:nnN #1#2#3
  {
    \group_begin:
      \__regex_multi_match:n { \__regex_extract: }
      #1
      \exp_args:No \__regex_match:n {#3}
      \exp_args:No \__regex_query_set:n {#3}
      #2
```

```
      \int_set:Nn \l__regex_balance_int
        {
          \c_zero_int
          \int_step_function:nnnN
            \l__regex_min_submatch_int
            \l__regex_capturing_group_int
            { \l__regex_submatch_int - \c_one_int }
            \__regex_replacement_balance_one_match:n
        }
      \__kernel_tl_set:Nx \l__regex_internal_a_tl
        {
          \int_step_function:nnnN
            \l__regex_min_submatch_int
            \l__regex_capturing_group_int
            { \l__regex_submatch_int - \c_one_int }
            \__regex_replacement_do_one_match:n
          \__regex_query_range:nn
            \l__regex_start_pos_int \l__regex_max_pos_int
        }
    \__regex_group_end_replace:N #3
  }
```

(*End of definition for* `\__regex_replace_all:nnN`*.*)

`\__regex_group_end_replace:N`
`\__regex_group_end_replace_try:`
`\__regex_group_end_replace_check:w`
`\__regex_group_end_replace_check:n`

At this stage `\l__regex_internal_a_tl` (e-expands to the desired result). Guess from `\l__regex_balance_int` the number of braces to add before or after the result then try expanding. The simplest case is when `\l__regex_internal_a_tl` together with the braces we insert via `\prg_replicate:nn` give a balanced result, and the assignment ends at the `\if_false: { \fi: }` construction: then `\__regex_group_end_replace_check:w` sees that there is no material left and we successfully found the result. The harder case is that expanding `\l__regex_internal_a_tl` may produce extra closing braces and end the assignment early. Then we grab the remaining code using; importantly, what follows has not yet been expanded so that `\__regex_group_end_replace_check:n` grabs everything until the last brace in `\__regex_group_end_replace_try:`, letting us try again with an extra surrounding pair of braces.

```
\cs_new_protected:Npn \__regex_group_end_replace:N #1
  {
    \int_set:Nn \l__regex_added_begin_int
      { \int_max:nn { - \l__regex_balance_int } \c_zero_int }
    \int_set:Nn \l__regex_added_end_int
      { \int_max:nn \l__regex_balance_int \c_zero_int }
    \__regex_group_end_replace_try:
    \int_compare:nNnT { \l__regex_added_begin_int + \l__regex_added_end_int }
                    > \c_zero_int
      {
        \msg_error:nneee { regex } { result-unbalanced }
          { replacing } { \int_use:N \l__regex_added_begin_int }
          { \int_use:N \l__regex_added_end_int }
      }
    \group_end:
    \tl_set_eq:NN #1 \g__regex_internal_tl
  }
\cs_new_protected:Npn \__regex_group_end_replace_try:
```

```
  {
    \tex_afterassignment:D \__regex_group_end_replace_check:w
    \__kernel_tl_gset:Nx \g__regex_internal_tl
      {
        \prg_replicate:nn \l__regex_added_begin_int { { \if_false: } \fi: }
        \l__regex_internal_a_tl
        \prg_replicate:nn \l__regex_added_end_int { \if_false: { \fi: } }
        \if_false: { \fi: }
      }
  }
\cs_new_protected:Npn \__regex_group_end_replace_check:w
  {
    \exp_after:wN \__regex_group_end_replace_check:n
    \exp_after:wN { \if_false: } \fi:
  }
\cs_new_protected:Npn \__regex_group_end_replace_check:n #1
  {
    \tl_if_empty:nF {#1}
      {
        \int_incr:N \l__regex_added_begin_int
        \int_incr:N \l__regex_added_end_int
        \__regex_group_end_replace_try:
      }
  }
```

(*End of definition for* `\__regex_group_end_replace:N` *and others.*)

46.7.5 Peeking ahead

`\l__regex_peek_true_tl`
`\l__regex_peek_false_tl`

True/false code arguments of `\peek_regex:nTF` or similar.

```
\tl_new:N \l__regex_peek_true_tl
\tl_new:N \l__regex_peek_false_tl
```

(*End of definition for* `\l__regex_peek_true_tl` *and* `\l__regex_peek_false_tl`*.*)

`\l__regex_replacement_tl`

When peeking in `\peek_regex_replace_once:nnTF` we need to store the replacement text.

```
\tl_new:N \l__regex_replacement_tl
```

(*End of definition for* `\l__regex_replacement_tl`*.*)

`\l__regex_input_tl`
`\__regex_input_item:n`

Stores each token found as `\__regex_input_item:n {⟨tokens⟩}`, where the ⟨*tokens*⟩ o-expand to the token found, as for `\tl_analysis_map_inline:nn`.

```
\tl_new:N \l__regex_input_tl
\cs_new_eq:NN \__regex_input_item:n ?
```

(*End of definition for* `\l__regex_input_tl` *and* `\__regex_input_item:n`*.*)

`\peek_regex:nTF`
`\peek_regex:NTF`
`\peek_regex_remove_once:nTF`
`\peek_regex_remove_once:NTF`

The `T` and `F` functions just call the corresponding `TF` function. The four `TF` functions differ along two axes: whether to remove the token or not, distinguished by using `\__regex_peek_end:` or `\__regex_peek_remove_end:n` (the latter case needs an argument, as we will see), and whether the regex has to be compiled or is already in an `N`-type variable, distinguished by calling `\__regex_build_aux:Nn` or `\__regex_build_aux:NN`. The first argument of these functions is `\c_false_bool` to indicate that there should be no implicit insertion of a wildcard at the start of the pattern: otherwise the code would keep looking further into the input stream until matching the regex.

```
\cs_new_protected:Npn \peek_regex:nTF #1
  {
    \__regex_peek:nnTF
      { \__regex_build_aux:Nn \c_false_bool {#1} }
      { \__regex_peek_end: }
  }
\cs_new_protected:Npn \peek_regex:nT #1#2
  { \peek_regex:nTF {#1} {#2} { } }
\cs_new_protected:Npn \peek_regex:nF #1 { \peek_regex:nTF {#1} { } }
\cs_new_protected:Npn \peek_regex:NTF #1
  {
    \__regex_peek:nnTF
      { \__regex_build_aux:NN \c_false_bool #1 }
      { \__regex_peek_end: }
  }
\cs_new_protected:Npn \peek_regex:NT #1#2
  { \peek_regex:NTF #1 {#2} { } }
\cs_new_protected:Npn \peek_regex:NF #1 { \peek_regex:NTF {#1} { } }
\cs_new_protected:Npn \peek_regex_remove_once:nTF #1
  {
    \__regex_peek:nnTF
      { \__regex_build_aux:Nn \c_false_bool {#1} }
      { \__regex_peek_remove_end:n {##1} }
  }
\cs_new_protected:Npn \peek_regex_remove_once:nT #1#2
  { \peek_regex_remove_once:nTF {#1} {#2} { } }
\cs_new_protected:Npn \peek_regex_remove_once:nF #1
  { \peek_regex_remove_once:nTF {#1} { } }
\cs_new_protected:Npn \peek_regex_remove_once:NTF #1
  {
    \__regex_peek:nnTF
      { \__regex_build_aux:NN \c_false_bool #1 }
      { \__regex_peek_remove_end:n {##1} }
  }
\cs_new_protected:Npn \peek_regex_remove_once:NT #1#2
  { \peek_regex_remove_once:NTF #1 {#2} { } }
\cs_new_protected:Npn \peek_regex_remove_once:NF #1
  { \peek_regex_remove_once:NTF #1 { } }
```

(*End of definition for* `\peek_regex:nTF` *and others. These functions are documented on page 208.*)

`\__regex_peek:nnTF`
`\__regex_peek_aux:nnTF`

Store the user's true/false codes (plus `\group_end:`) into two token lists. Then build the automaton with `#1`, without submatch tracking, and aiming for a single match. Then start matching by setting up a few variables like for any regex matching like `\regex_match:nnTF`, with the addition of `\l__regex_input_tl` that keeps track of the tokens seen, to reinsert them at the end. Instead of `\tl_analysis_map_inline:nn` on the input, we call `\peek_analysis_map_inline:n` to go through tokens in the input stream. Since `\__regex_match_one_token:nnN` calls `\__regex_maplike_break:` we need to catch that and break the `\peek_analysis_map_inline:n` loop instead.

```
\cs_new_protected:Npn \__regex_peek:nnTF #1
  {
```

```
    \__regex_peek_aux:nnTF
      {
        \__regex_disable_submatches:
        #1
      }
  }
\cs_new_protected:Npn \__regex_peek_aux:nnTF #1#2#3#4
  {
    \group_begin:
      \tl_set:Nn \l__regex_peek_true_tl { \group_end: #3 }
      \tl_set:Nn \l__regex_peek_false_tl { \group_end: #4 }
      \__regex_single_match:
      #1
      \__regex_match_init:
      \tl_build_begin:N \l__regex_input_tl
      \__regex_match_once_init:
      \peek_analysis_map_inline:n
        {
          \tl_build_put_right:Nn \l__regex_input_tl
            { \__regex_input_item:n {##1} }
          \__regex_match_one_token:nnN {##1} {##2} ##3
          \use_none:nnn
          \prg_break_point:Nn \__regex_maplike_break:
            { \peek_analysis_map_break:n {#2} }
        }
  }
```

(*End of definition for* `\__regex_peek:nnTF` *and* `\__regex_peek_aux:nnTF`.)

`\__regex_peek_end:`
`\__regex_peek_remove_end:n`

Once the regex matches (or permanently fails to match) we call `\__regex_peek_end:`, or `\__regex_peek_remove_end:n` with argument the last token seen. For `\peek_regex:nTF` we reinsert tokens seen by calling `\__regex_peek_reinsert:N` regardless of the result of the match. For `\peek_regex_remove_once:nTF` we reinsert the tokens seen only if the match failed; otherwise we just reinsert the tokens `#1`, with one expansion. To be more precise, `#1` consists of tokens that `o`-expand and `e`-expand to the last token seen, for example it is `\exp_not:N` ⟨*cs*⟩ for a control sequence. This means that just doing `\exp_after:wN \l__regex_peek_true_tl #1` would be unsafe because the expansion of ⟨*cs*⟩ would be suppressed.

```
\cs_new_protected:Npn \__regex_peek_end:
  {
    \bool_if:NTF \g__regex_success_bool
      { \__regex_peek_reinsert:N \l__regex_peek_true_tl }
      { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
  }
\cs_new_protected:Npn \__regex_peek_remove_end:n #1
  {
    \bool_if:NTF \g__regex_success_bool
      { \exp_args:NNo \use:nn \l__regex_peek_true_tl {#1} }
      { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
  }
```

(*End of definition for* `\__regex_peek_end:` *and* `\__regex_peek_remove_end:n`.)

__regex_peek_reinsert:N
__regex_reinsert_item:n

Insert the true/false code #1, followed by the tokens found, which were stored in \l__regex_input_tl. For this, loop through that token list using __regex_reinsert_item:n, which expands #1 once to get a single token, and jumps over it to expand what follows, with suitable \exp:w and \exp_end:. We cannot just use \use:e on the whole token list because the result may be unbalanced, which would stop the primitive prematurely, or let it continue beyond where we would like.

```
\cs_new_protected:Npn \__regex_peek_reinsert:N #1
  {
    \tl_build_end:N \l__regex_input_tl
    \cs_set_eq:NN \__regex_input_item:n \__regex_reinsert_item:n
    \exp_after:wN #1 \exp:w \l__regex_input_tl \exp_end:
  }
\cs_new_protected:Npn \__regex_reinsert_item:n #1
  {
    \exp_after:wN \exp_after:wN
    \exp_after:wN \exp_end:
    \exp_after:wN \exp_after:wN
    #1
    \exp:w
  }
```

(*End of definition for __regex_peek_reinsert:N and __regex_reinsert_item:n.*)

\peek_regex_replace_once:nn
\peek_regex_replace_once:nnTF
\peek_regex_replace_once:Nn
\peek_regex_replace_once:NnTF

Similar to \peek_regex:nTF above.

```
\cs_new_protected:Npn \peek_regex_replace_once:nnTF #1
  { \__regex_peek_replace:nnTF { \__regex_build_aux:Nn \c_false_bool {#1} } }
\cs_new_protected:Npn \peek_regex_replace_once:nnT #1#2#3
  { \peek_regex_replace_once:nnTF {#1} {#2} {#3} { } }
\cs_new_protected:Npn \peek_regex_replace_once:nnF #1#2
  { \peek_regex_replace_once:nnTF {#1} {#2} { } }
\cs_new_protected:Npn \peek_regex_replace_once:nn #1#2
  { \peek_regex_replace_once:nnTF {#1} {#2} { } { } }
\cs_new_protected:Npn \peek_regex_replace_once:NnTF #1
  { \__regex_peek_replace:nnTF { \__regex_build_aux:NN \c_false_bool #1 } }
\cs_new_protected:Npn \peek_regex_replace_once:NnT #1#2#3
  { \peek_regex_replace_once:NnTF #1 {#2} {#3} { } }
\cs_new_protected:Npn \peek_regex_replace_once:NnF #1#2
  { \peek_regex_replace_once:NnTF #1 {#2} { } }
\cs_new_protected:Npn \peek_regex_replace_once:Nn #1#2
  { \peek_regex_replace_once:NnTF #1 {#2} { } { } }
```

(*End of definition for \peek_regex_replace_once:nnTF and \peek_regex_replace_once:NnTF. These functions are documented on page* *209**.*)

__regex_peek_replace:nnTF

Same as __regex_peek:nnTF (used for \peek_regex:nTF above), but without disabling submatches, and with a different end. The replacement text #2 is stored, to be analyzed later.

```
\cs_new_protected:Npn \__regex_peek_replace:nnTF #1#2
  {
    \tl_set:Nn \l__regex_replacement_tl {#2}
    \__regex_peek_aux:nnTF {#1} { \__regex_peek_replace_end: }
  }
```

(*End of definition for __regex_peek_replace:nnTF.*)

`\__regex_peek_replace_end:` If the match failed `\__regex_peek_reinsert:N` reinserts the tokens found. Otherwise, finish storing the submatch information using `\__regex_extract:`, and store the input into `\toks`. Redefine a few auxiliaries to change slightly their expansion behaviour as explained below. Analyse the replacement text with `\__regex_replacement:n`, which as usual defines `\__regex_replacement_do_one_match:n` to insert the tokens from the start of the match attempt to the beginning of the match, followed by the replacement text. The `\use:e` expands for instance the trailing `\__regex_query_range:nn` down to a sequence of `\__regex_reinsert_item:n {⟨tokens⟩}` where ⟨*tokens*⟩ o-expand to a single token that we want to insert. After e-expansion, `\use:e` does `\use:n`, so we have `\exp_after:wN \l__regex_peek_true_tl \exp:w ... \exp_end:`. This is set up such as to obtain `\l__regex_peek_true_tl` followed by the replaced tokens (possibly unbalanced) in the input stream.

```
\cs_new_protected:Npn \__regex_peek_replace_end:
  {
    \bool_if:NTF \g__regex_success_bool
      {
        \__regex_extract:
        \__regex_query_set_from_input_tl:
        \cs_set_eq:NN \__regex_replacement_put:n \__regex_peek_replacement_put:n
        \cs_set_eq:NN \__regex_replacement_put_submatch_aux:n
          \__regex_peek_replacement_put_submatch_aux:n
        \cs_set_eq:NN \__regex_input_item:n \__regex_reinsert_item:n
        \cs_set_eq:NN \__regex_replacement_exp_not:N \__regex_peek_replacement_token:n
        \cs_set_eq:NN \__regex_replacement_exp_not:V \__regex_peek_replacement_var:N
        \exp_args:No \__regex_replacement:n { \l__regex_replacement_tl }
        \use:e
          {
            \exp_not:n { \exp_after:wN \l__regex_peek_true_tl \exp:w }
            \__regex_replacement_do_one_match:n \l__regex_zeroth_submatch_int
            \__regex_query_range:nn
              {
                \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
                  \l__regex_zeroth_submatch_int
              }
              \l__regex_max_pos_int
            \exp_end:
          }
      }
      { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
  }
```

(*End of definition for* `\__regex_peek_replace_end:`*.*)

`\__regex_query_set_from_input_tl:` `\__regex_query_set_item:n` The input was stored into `\l__regex_input_tl` as successive items `\__regex_input_item:n {⟨tokens⟩}`. Store that in successive `\toks`. It's not clear whether the empty entries before and after are both useful.

```
\cs_new_protected:Npn \__regex_query_set_from_input_tl:
  {
    \tl_build_end:N \l__regex_input_tl
    \int_zero:N \l__regex_curr_pos_int
    \cs_set_eq:NN \__regex_input_item:n \__regex_query_set_item:n
    \__regex_query_set_item:n { }
    \l__regex_input_tl
```

```
    \__regex_query_set_item:n { }
    \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
  }
\cs_new_protected:Npn \__regex_query_set_item:n #1
  {
    \int_incr:N \l__regex_curr_pos_int
    \__regex_toks_set:Nn \l__regex_curr_pos_int { \__regex_input_item:n {#1} }
  }
```

(*End of definition for* `\__regex_query_set_from_input_tl:` *and* `\__regex_query_set_item:n`.)

`\__regex_peek_replacement_put:n`

While building the replacement function `\__regex_replacement_do_one_match:n`, we often want to put simple material, given as `#1`, whose `e`-expansion `o`-expands to a single token. Normally we can just add the token to `\l__regex_build_tl`, but for `\peek_regex_replace_once:nnTF` we eventually want to do some strange expansion that is basically using `\exp_after:wN` to jump through numerous tokens (we cannot use `e`-expansion like for `\regex_replace_once:nnNTF` because it is ok for the result to be unbalanced since we insert it in the input stream rather than storing it. When within a csname we don't do any such shenanigan because `\cs:w` … `\cs_end:` does all the expansion we need.

```
\cs_new_protected:Npn \__regex_peek_replacement_put:n #1
  {
    \if_case:w \l__regex_replacement_csnames_int
      \tl_build_put_right:Nn \l__regex_build_tl
        { \exp_not:N \__regex_reinsert_item:n {#1} }
    \else:
      \tl_build_put_right:Nn \l__regex_build_tl {#1}
    \fi:
  }
```

(*End of definition for* `\__regex_peek_replacement_put:n`.)

`\__regex_peek_replacement_token:n`

When hit with `\exp:w`, `\__regex_peek_replacement_token:n` {⟨*token*⟩} stops `\exp_end:` and does `\exp_after:wN` ⟨*token*⟩ `\exp:w` to continue expansion after it.

```
\cs_new_protected:Npn \__regex_peek_replacement_token:n #1
  { \exp_after:wN \exp_end: \exp_after:wN #1 \exp:w }
```

(*End of definition for* `\__regex_peek_replacement_token:n`.)

`\__regex_peek_replacement_put_submatch_aux:n`

While analyzing the replacement we also have to insert submatches found in the query. Since query items `\__regex_input_item:n` {⟨*tokens*⟩} expand correctly only when surrounded by `\exp:w` … `\exp_end:`, and since these expansion controls are not there within csnames (because `\cs:w` … `\cs_end:` make them unnecessary in most cases), we have to put `\exp:w` and `\exp_end:` by hand here.

```
\cs_new_protected:Npn \__regex_peek_replacement_put_submatch_aux:n #1
  {
    \if_case:w \l__regex_replacement_csnames_int
      \tl_build_put_right:Nn \l__regex_build_tl
        { \__regex_query_submatch:n { \__regex_int_eval:w #1 + ##1 \scan_stop: } }
    \else:
      \tl_build_put_right:Nn \l__regex_build_tl
        {
          \exp:w
            \__regex_query_submatch:n { \__regex_int_eval:w #1 + ##1 \scan_stop: }
```

```
          \exp_end:
        }
    \fi:
  }
```

(*End of definition for* `\__regex_peek_replacement_put_submatch_aux:n`.)

`\__regex_peek_replacement_var:N`

This is used for `\u` outside csnames. It makes sure to continue expansion with `\exp:w` before expanding the variable `#1` and stopping the `\exp:w` that precedes.

```
\cs_new_protected:Npn \__regex_peek_replacement_var:N #1
  {
    \exp_after:wN \exp_last_unbraced:NV
    \exp_after:wN \exp_end:
    \exp_after:wN #1
    \exp:w
  }
```

(*End of definition for* `\__regex_peek_replacement_var:N`.)

46.8 Messages

Messages for the preparsing phase.

```
\use:e
  {
    \msg_new:nnn { regex } { trailing-backslash }
      { Trailing~'\iow_char:N\\'~in~regex~or~replacement. }
    \msg_new:nnn { regex } { x-missing-rbrace }
      {
        Missing~brace~'\iow_char:N\}'~in~regex~
        '...\iow_char:N\\x\iow_char:N\{...##1'.
      }
    \msg_new:nnn { regex } { x-overflow }
      {
        Character~code~##1~too~large~in~
        \iow_char:N\\x\iow_char:N\{##2\iow_char:N\}~regex.
      }
  }
```

Invalid quantifier.

```
\msg_new:nnnn { regex } { invalid-quantifier }
  { Braced~quantifier~'#1'~may~not~be~followed~by~'#2'. }
  {
    The~character~'#2'~is~invalid~in~the~braced~quantifier~'#1'.~
    The~only~valid~quantifiers~are~'*',~'?',~'+',~'{<int>}',~
    '{<min>,}'~and~'{<min>,<max>}',~optionally~followed~by~'?'.
  }
```

Messages for missing or extra closing brackets and parentheses, with some fancy singular/plural handling for the case of parentheses.

```
\msg_new:nnnn { regex } { missing-rbrack }
  { Missing~right~bracket~inserted~in~regular~expression. }
  {
    LaTeX~was~given~a~regular~expression~where~a~character~class~
    was~started~with~'[',~but~the~matching~']'~is~missing.
```

```
  }
\msg_new:nnnn { regex } { missing-rparen }
  {
    Missing~right~
    \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } ~
    inserted~in~regular~expression.
  }
  {
    LaTeX~was~given~a~regular~expression~with~\int_eval:n {#1} ~
    more~left~parentheses~than~right~parentheses.
  }
\msg_new:nnnn { regex } { extra-rparen }
  { Extra~right~parenthesis~ignored~in~regular~expression. }
  {
    LaTeX~came~across~a~closing~parenthesis~when~no~submatch~group~
    was~open.~The~parenthesis~will~be~ignored.
  }
```

Some escaped alphanumerics are not allowed everywhere.

```
\msg_new:nnnn { regex } { bad-escape }
  {
    Invalid~escape~'\iow_char:N\\#1'~
    \__regex_if_in_cs:TF { within~a~control~sequence. }
      {
        \__regex_if_in_class:TF
          { in~a~character~class. }
          { following~a~category~test. }
      }
  }
  {
    The~escape~sequence~'\iow_char:N\\#1'~may~not~appear~
    \__regex_if_in_cs:TF
      {
        within~a~control~sequence~test~introduced~by~
        '\iow_char:N\\c\iow_char:N\{'.
      }
      {
        \__regex_if_in_class:TF
          { within~a~character~class~ }
          { following~a~category~test~such~as~'\iow_char:N\\cL'~ }
        because~it~does~not~match~exactly~one~character.
      }
  }
```

Range errors.

```
\msg_new:nnnn { regex } { range-missing-end }
  { Invalid~end-point~for~range~'#1-#2'~in~character~class. }
  {
    The~end-point~'#2'~of~the~range~'#1-#2'~may~not~serve~as~an~
    end-point~for~a~range:~alphanumeric~characters~should~not~be~
    escaped,~and~non-alphanumeric~characters~should~be~escaped.
  }
\msg_new:nnnn { regex } { range-backwards }
  { Range~'[#1-#2]'~out~of~order~in~character~class. }
  {
```

```
    In~ranges~of~characters~'[x-y]'~appearing~in~character~classes,~
    the~first~character~code~must~not~be~larger~than~the~second.~
    Here,~'#1'~has~character~code~\int_eval:n {`#1},~while~
    '#2'~has~character~code~\int_eval:n {`#2}.
  }
```

Errors related to \c and \u.

```
\msg_new:nnnn { regex } { c-bad-mode }
  { Invalid~nested~'\iow_char:N\\c'~escape~in~regular~expression. }
  {
    The~'\iow_char:N\\c'~escape~cannot~be~used~within~
    a~control~sequence~test~'\iow_char:N\\c{...}'~
    nor~another~category~test.~
    To~combine~several~category~tests,~use~'\iow_char:N\\c[...]'.
  }
\msg_new:nnnn { regex } { c-C-invalid }
  { '\iow_char:N\\cC'~should~be~followed~by~'.'~or~'(',~not~'#1'. }
  {
    The~'\iow_char:N\\cC'~construction~restricts~the~next~item~to~be~a~
    control~sequence~or~the~next~group~to~be~made~of~control~sequences.~
    It~only~makes~sense~to~follow~it~by~'.'~or~by~a~group.
  }
\msg_new:nnnn { regex } { cu-lbrace }
  { Left~braces~must~be~escaped~in~'\iow_char:N\\#1{...}'. }
  {
    Constructions~such~as~'\iow_char:N\\#1{...\iow_char:N\{...}'~are~
    not~allowed~and~should~be~replaced~by~
    '\iow_char:N\\#1{...\token_to_str:N\{...}'.
  }
\msg_new:nnnn { regex } { c-lparen-in-class }
  { Catcode~test~cannot~apply~to~group~in~character~class }
  {
    Construction~such~as~'\iow_char:N\\cL(abc)'~are~not~allowed~inside~a~
    class~'[...]'~because~classes~do~not~match~multiple~characters~at~once.
  }
\msg_new:nnnn { regex } { c-missing-rbrace }
  { Missing~right~brace~inserted~for~'\iow_char:N\\c'~escape. }
  {
    LaTeX~was~given~a~regular~expression~where~a~
    '\iow_char:N\\c\iow_char:N\{...'~construction~was~not~ended~
    with~a~closing~brace~'\iow_char:N\}'.
  }
\msg_new:nnnn { regex } { c-missing-rbrack }
  { Missing~right~bracket~inserted~for~'\iow_char:N\\c'~escape. }
  {
    A~construction~'\iow_char:N\\c[...'~appears~in~a~
    regular~expression,~but~the~closing~']'~is~not~present.
  }
\msg_new:nnnn { regex } { c-missing-category }
  { Invalid~character~'#1'~following~'\iow_char:N\\c'~escape. }
  {
    In~regular~expressions,~the~'\iow_char:N\\c'~escape~sequence~
    may~only~be~followed~by~a~left~brace,~a~left~bracket,~or~a~
    capital~letter~representing~a~character~category,~namely~
    one~of~'ABCDELMOPSTU'.
```

```
  }
\msg_new:nnnn { regex } { c-trailing }
  { Trailing~category~code~escape~'\iow_char:N\\c'... }
  {
    A~regular~expression~ends~with~'\iow_char:N\\c'~followed~
    by~a~letter.~It~will~be~ignored.
  }
\msg_new:nnnn { regex } { u-missing-lbrace }
  { Missing~left~brace~following~'\iow_char:N\\u'~escape. }
  {
    The~'\iow_char:N\\u'~escape~sequence~must~be~followed~by~
    a~brace~group~with~the~name~of~the~variable~to~use.
  }
\msg_new:nnnn { regex } { u-missing-rbrace }
  { Missing~right~brace~inserted~for~'\iow_char:N\\u'~escape. }
  {
    LaTeX~
    \str_if_eq:eeTF { } {#2}
      { reached~the~end~of~the~string~ }
      { encountered~an~escaped~alphanumeric~character '\iow_char:N\\#2'~ }
    when~parsing~the~argument~of~an~
    '\iow_char:N\\u\iow_char:N\{...\}'~escape.
  }
```

Errors when encountering the POSIX syntax [:...:].

```
\msg_new:nnnn { regex } { posix-unsupported }
  { POSIX~collating~element~'[#1 ~ #1]'~not~supported. }
  {
    The~'[.foo.]'~and~'[=bar=]'~syntaxes~have~a~special~meaning~
    in~POSIX~regular~expressions.~This~is~not~supported~by~LaTeX.~
    Maybe~you~forgot~to~escape~a~left~bracket~in~a~character~class?
  }
\msg_new:nnnn { regex } { posix-unknown }
  { POSIX~class~'[:#1:]'~unknown. }
  {
    '[:#1:]'~is~not~among~the~known~POSIX~classes~
    '[:alnum:]',~'[:alpha:]',~'[:ascii:]',~'[:blank:]',~
    '[:cntrl:]',~'[:digit:]',~'[:graph:]',~'[:lower:]',~
    '[:print:]',~'[:punct:]',~'[:space:]',~'[:upper:]',~
    '[:word:]',~and~'[:xdigit:]'.
  }
\msg_new:nnnn { regex } { posix-missing-close }
  { Missing~closing~':]'~for~POSIX~class. }
  { The~POSIX~syntax~'#1'~must~be~followed~by~':]',~not~'#2'. }
```

In various cases, the result of a l3regex operation can leave us with an unbalanced token list, which we must re-balance by adding begin-group or end-group character tokens.

```
\msg_new:nnnn { regex } { result-unbalanced }
  { Missing~brace~inserted~when~#1. }
  {
    LaTeX~was~asked~to~do~some~regular~expression~operation,~
    and~the~resulting~token~list~would~not~have~the~same~number~
    of~begin-group~and~end-group~tokens.~Braces~were~inserted:~
    #2~left,~#3~right.
```

```
  }
```

Error message for unknown options.

```
\msg_new:nnnn { regex } { unknown-option }
  { Unknown~option~'#1'~for~regular~expressions. }
  {
    The~only~available~option~is~'case-insensitive',~toggled~by~
    '(?i)'~and~'(?-i)'.
  }
\msg_new:nnnn { regex } { special-group-unknown }
  { Unknown~special~group~'#1~...'~in~a~regular~expression. }
  {
    The~only~valid~constructions~starting~with~'(?'~are~
    '(?:~...~)',~'(?|~...~)',~'(?i)',~and~'(?-i)'.
  }
```

Errors in the replacement text.

```
\msg_new:nnnn { regex } { replacement-c }
  { Misused~'\iow_char:N\\c'~command~in~a~replacement~text. }
  {
    In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
    can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~
    or~a~brace~group,~not~by~'#1'.
  }
\msg_new:nnnn { regex } { replacement-u }
  { Misused~'\iow_char:N\\u'~command~in~a~replacement~text. }
  {
    In~a~replacement~text,~the~'\iow_char:N\\u'~escape~sequence~
    must~be~~followed~by~a~brace~group~holding~the~name~of~the~
    variable~to~use.
  }
\msg_new:nnnn { regex } { replacement-g }
  {
    Missing~brace~for~the~'\iow_char:N\\g'~construction~
    in~a~replacement~text.
  }
  {
    In~the~replacement~text~for~a~regular~expression~search,~
    submatches~are~represented~either~as~'\iow_char:N \\g{dd..d}',~
    or~'\\d',~where~'d'~are~single~digits.~Here,~a~brace~is~missing.
  }
\msg_new:nnnn { regex } { replacement-catcode-end }
  {
    Missing~character~for~the~'\iow_char:N\\c<category><character>'~
    construction~in~a~replacement~text.
  }
  {
    In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
    can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
    the~character~category.~Then,~a~character~must~follow.~LaTeX~
    reached~the~end~of~the~replacement~when~looking~for~that.
  }
\msg_new:nnnn { regex } { replacement-catcode-escaped }
  {
    Escaped~letter~or~digit~after~category~code~in~replacement~text.
```

```
  }
  {
    In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
    can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
    the~character~category.~Then,~a~character~must~follow,~not~
    '\iow_char:N\\#2'.
  }
\msg_new:nnnn { regex } { replacement-catcode-in-cs }
  {
    Category~code~'\iow_char:N\\c#1#3'~ignored~inside~
    '\iow_char:N\\c\{...\}'~in~a~replacement~text.
  }
  {
    In~a~replacement~text,~the~category~codes~of~the~argument~of~
    '\iow_char:N\\c\{...\}'~are~ignored~when~building~the~control~
    sequence~name.
  }
\msg_new:nnnn { regex } { replacement-null-space }
  { TeX~cannot~build~a~space~token~with~character~code~0. }
  {
    You~asked~for~a~character~token~with~category~space,~
    and~character~code~0,~for~instance~through~
    '\iow_char:N\\cS\iow_char:N\\x00'.~
    This~specific~case~is~impossible~and~will~be~replaced~
    by~a~normal~space.
  }
\msg_new:nnnn { regex } { replacement-missing-rbrace }
  { Missing~right~brace~inserted~in~replacement~text. }
  {
    There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
    missing~right~\int_compare:nTF { #1 = 1 } { brace } { braces } .
  }
\msg_new:nnnn { regex } { replacement-missing-rparen }
  { Missing~right~parenthesis~inserted~in~replacement~text. }
  {
    There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
    missing~right~
    \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } .
  }
\msg_new:nnn { regex } { submatch-too-big }
  { Submatch~#1~used~but~regex~only~has~#2~group(s) }
```

Some escaped alphanumerics are not allowed everywhere.

```
\msg_new:nnnn { regex } { backwards-quantifier }
  { Quantifer~"{#1,#2}"~is~backwards. }
  { The~values~given~in~a~quantifier~must~be~in~order. }
```

Used in user commands, and when showing a regex.

```
\msg_new:nnnn { regex } { case-odd }
  { #1~with~odd~number~of~items }
  {
    There~must~be~a~#2~part~for~each~regex:~
    found~odd~number~of~items~(#3)~in\\
    \iow_indent:n {#4}
  }
```

```
\msg_new:nnn { regex } { show }
  {
    >~Compiled~regex~
    \tl_if_empty:nTF {#1} { variable~ #2 } { {#1} } :
    #3
  }
\prop_gput:Nnn \g_msg_module_name_prop { regex } { LaTeX }
\prop_gput:Nnn \g_msg_module_type_prop { regex } { }
```

__regex_msg_repeated:nnN This is not technically a message, but seems related enough to go there. The arguments are: #1 is the minimum number of repetitions; #2 is the number of allowed extra repetitions (-1 for infinite number), and #3 tells us about laziness.

```
\cs_new:Npn \__regex_msg_repeated:nnN #1#2#3
  {
    \str_if_eq:eeF { #1 #2 } { 1 0 }
      {
        , ~ repeated ~
        \int_case:nnF {#2}
          {
            { -1 } { #1~or~more~times,~\bool_if:NTF #3 { lazy } { greedy } }
            {  0 } { #1~times }
          }
          {
            between~#1~and~\int_eval:n {#1+#2}~times,~
            \bool_if:NTF #3 { lazy } { greedy }
          }
      }
  }
```

(*End of definition for* __regex_msg_repeated:nnN.)

46.9 Code for tracing

There is a more extensive implementation of tracing in the l3trial package l3trace. Function names are a bit different but could be merged.

__regex_trace_push:nnN
__regex_trace_pop:nnN
__regex_trace:nne

Here #1 is the module name (regex) and #2 is typically 1. If the module's current tracing level is less than #2 show nothing, otherwise write #3 to the terminal.

```
\cs_new_protected:Npn \__regex_trace_push:nnN #1#2#3
  { \__regex_trace:nne {#1} {#2} { entering~ \token_to_str:N #3 } }
\cs_new_protected:Npn \__regex_trace_pop:nnN #1#2#3
  { \__regex_trace:nne {#1} {#2} { leaving~ \token_to_str:N #3 } }
\cs_new_protected:Npn \__regex_trace:nne #1#2#3
  {
    \int_compare:nNnF
      { \int_use:c { g__regex_trace_#1_int } } < {#2}
      { \iow_term:e { Trace:~#3 } }
  }
```

(*End of definition for* __regex_trace_push:nnN, __regex_trace_pop:nnN, *and* __regex_trace:nne.)

\g__regex_trace_regex_int No tracing when that is zero.

```
\int_new:N \g__regex_trace_regex_int
```

(*End of definition for* `\g__regex_trace_regex_int`*.*)

`\__regex_trace_states:n` This function lists the contents of all states of the NFA, stored in `\toks` from 0 to `\l__regex_max_state_int` (excluded).

```
\cs_new_protected:Npn \__regex_trace_states:n #1
  {
    \int_step_inline:nnn
      \l__regex_min_state_int
      { \l__regex_max_state_int - \c_one_int }
      {
        \__regex_trace:nne { regex } {#1}
          { \iow_char:N \\toks ##1 = { \__regex_toks_use:w ##1 } }
      }
  }
```

(*End of definition for* `\__regex_trace_states:n`*.*)

```
⟨/package⟩
```

Chapter 47

l3prg implementation

The following test files are used for this code: m3prg001.lvt,m3prg002.lvt,m3prg003.lvt.

```
⟨*package⟩
```

47.1 Primitive conditionals

\if_bool:N
\if_predicate:w

Those two primitive TeX conditionals are synonyms. `\if_bool:N` is defined in l3basics, as it's needed earlier to define quark test functions.

```
\cs_new_eq:NN \if_predicate:w \tex_ifodd:D
```

(*End of definition for* `\if_bool:N` *and* `\if_predicate:w`. *These functions are documented on page 73.*)

47.2 Defining a set of conditional functions

\prg_set_conditional:Npnn
\prg_new_conditional:Npnn
\prg_set_protected_conditional:Npnn
\prg_new_protected_conditional:Npnn
\prg_set_conditional:Nnn
\prg_new_conditional:Nnn
\prg_set_protected_conditional:Nnn
\prg_new_protected_conditional:Nnn
\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn
\prg_return_true:
\prg_return_false:

These are all defined in l3basics, as they are needed "early". This is just a reminder!

(*End of definition for* `\prg_set_conditional:Npnn` *and others. These functions are documented on page 64.*)

47.3 The boolean data type

```
⟨@@=bool⟩
```

\bool_new:N
\bool_new:c

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

```
\cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
\cs_generate_variant:Nn \bool_new:N { c }
```

(*End of definition for* `\bool_new:N`. *This function is documented on page 67.*)

\bool_const:Nn
\bool_const:cn

A merger between `\tl_const:Nn` and `\bool_set:Nn`.

```
\cs_new_protected:Npn \bool_const:Nn #1#2
  {
    \__kernel_chk_if_free_cs:N #1
    \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}
  }
\cs_generate_variant:Nn \bool_const:Nn { c }
```

(*End of definition for* `\bool_const:Nn`. *This function is documented on page 67.*)

`\bool_set_true:N`
`\bool_set_true:c`
`\bool_gset_true:N`
`\bool_gset_true:c`
`\bool_set_false:N`
`\bool_set_false:c`
`\bool_gset_false:N`
`\bool_gset_false:c`

Setting is already pretty easy. When `check-declarations` is active, the definitions are patched to make sure the boolean exists. This is needed because booleans are not based on token lists nor on TeX registers.

```
\cs_new_protected:Npn \bool_set_true:N #1
  { \cs_set_eq:NN #1 \c_true_bool }
\cs_new_protected:Npn \bool_set_false:N #1
  { \cs_set_eq:NN #1 \c_false_bool }
\cs_new_protected:Npn \bool_gset_true:N #1
  { \cs_gset_eq:NN #1 \c_true_bool }
\cs_new_protected:Npn \bool_gset_false:N #1
  { \cs_gset_eq:NN #1 \c_false_bool }
\cs_generate_variant:Nn \bool_set_true:N   { c }
\cs_generate_variant:Nn \bool_set_false:N  { c }
\cs_generate_variant:Nn \bool_gset_true:N  { c }
\cs_generate_variant:Nn \bool_gset_false:N { c }
```

(*End of definition for* `\bool_set_true:N` *and others. These functions are documented on page 67.*)

`\bool_set_eq:NN`
`\bool_set_eq:cN`
`\bool_set_eq:Nc`
`\bool_set_eq:cc`
`\bool_gset_eq:NN`
`\bool_gset_eq:cN`
`\bool_gset_eq:Nc`
`\bool_gset_eq:cc`

The usual copy code. While it would be cleaner semantically to copy the `\cs_set_eq:NN` family of functions, we copy `\tl_set_eq:NN` because that has the correct checking code.

```
\cs_new_eq:NN \bool_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \bool_gset_eq:NN \tl_gset_eq:NN
\cs_generate_variant:Nn \bool_set_eq:NN { Nc, cN, cc }
\cs_generate_variant:Nn \bool_gset_eq:NN { Nc, cN, cc }
```

(*End of definition for* `\bool_set_eq:NN` *and* `\bool_gset_eq:NN`. *These functions are documented on page 67.*)

`\bool_set:Nn`
`\bool_set:cn`
`\bool_gset:Nn`
`\bool_gset:cn`

This function evaluates a boolean expression and assigns the first argument the meaning `\c_true_bool` or `\c_false_bool`. Again, we include some checking code. It is important to evaluate the expression before applying the `\chardef` primitive, because that primitive sets the left-hand side to `\scan_stop:` before looking for the right-hand side.

```
\cs_new_protected:Npn \bool_set:Nn #1#2
  {
    \exp_last_unbraced:NNNf
      \tex_chardef:D #1 = { \bool_if_p:n {#2} }
  }
\cs_new_protected:Npn \bool_gset:Nn #1#2
  {
    \exp_last_unbraced:NNNNf
      \tex_global:D \tex_chardef:D #1 = { \bool_if_p:n {#2} }
  }
\cs_generate_variant:Nn \bool_set:Nn  { c }
\cs_generate_variant:Nn \bool_gset:Nn { c }
```

(*End of definition for* `\bool_set:Nn` *and* `\bool_gset:Nn`. *These functions are documented on page 67.*)

`\bool_set_inverse:N`
`\bool_set_inverse:c`
`\bool_gset_inverse:N`
`\bool_gset_inverse:c`

Set to `false` or `true` locally or globally.

```
\cs_new_protected:Npn \bool_set_inverse:N #1
  { \bool_if:NTF #1 { \bool_set_false:N } { \bool_set_true:N } #1 }
\cs_generate_variant:Nn \bool_set_inverse:N { c }
\cs_new_protected:Npn \bool_gset_inverse:N #1
  { \bool_if:NTF #1 { \bool_gset_false:N } { \bool_gset_true:N } #1 }
\cs_generate_variant:Nn \bool_gset_inverse:N { c }
```

(*End of definition for* \bool_set_inverse:N *and* \bool_gset_inverse:N*. These functions are documented on page* 67.)

47.4 Internal auxiliaries

\q__bool_recursion_tail
\q__bool_recursion_stop

Internal recursion quarks.

```
\quark_new:N \q__bool_recursion_tail
\quark_new:N \q__bool_recursion_stop
```

(*End of definition for* \q__bool_recursion_tail *and* \q__bool_recursion_stop*.*)

__bool_use_i_delimit_by_q_recursion_stop:nw

Functions to gobble up to a quark.

```
\cs_new:Npn \__bool_use_i_delimit_by_q_recursion_stop:nw
  #1 #2 \q__bool_recursion_stop {#1}
```

(*End of definition for* __bool_use_i_delimit_by_q_recursion_stop:nw*.*)

__bool_if_recursion_tail_stop_do:nn

Functions to query recursion quarks.

```
\__kernel_quark_new_test:N \__bool_if_recursion_tail_stop_do:nn
```

(*End of definition for* __bool_if_recursion_tail_stop_do:nn*.*)

\bool_if_p:N
\bool_if_p:c
\bool_if:N*TF*
\bool_if:c*TF*

Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

```
\prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
  {
    \if_bool:N #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_generate_conditional_variant:Nnn \bool_if:N { c } { p , T , F , TF }
```

(*End of definition for* \bool_if:NTF*. This function is documented on page* 68.)

\bool_to_str:N
\bool_to_str:c
\bool_to_str:n

Expands to string literal `true` or `false`.

```
\cs_new:Npe \bool_to_str:N #1
  {
    \exp_not:N \bool_if:NTF #1
      { \tl_to_str:n { true } } { \tl_to_str:n { false } }
  }
\cs_generate_variant:Nn \bool_to_str:N { c }
\cs_new:Npe \bool_to_str:n #1
  {
    \exp_not:N \bool_if:nTF {#1}
      { \tl_to_str:n { true } } { \tl_to_str:n { false } }
  }
```

(*End of definition for* \bool_to_str:N *and* \bool_to_str:n*. These functions are documented on page* 68.)

\bool_show:n
\bool_log:n

Show the truth value of the boolean.

```
\cs_new_protected:Npn \bool_show:n
  { \__kernel_msg_show_eval:Nn \bool_to_str:n }
\cs_new_protected:Npn \bool_log:n
  { \__kernel_msg_log_eval:Nn \bool_to_str:n }
```

(*End of definition for* `\bool_show:n` *and* `\bool_log:n`*. These functions are documented on page 68.*)

\bool_show:N
\bool_show:c
\bool_log:N
\bool_log:c
__bool_show:NN

Show the truth value of the boolean, as `true` or `false`.

```
\cs_new_protected:Npn \bool_show:N { \__bool_show:NN \tl_show:n }
\cs_generate_variant:Nn \bool_show:N { c }
\cs_new_protected:Npn \bool_log:N { \__bool_show:NN \tl_log:n }
\cs_generate_variant:Nn \bool_log:N { c }
\cs_new_protected:Npn \__bool_show:NN #1#2
  {
    \__kernel_chk_defined:NT #2
      {
        \token_case_meaning:NnF #2
          {
            \c_true_bool { \exp_args:Ne #1 { \token_to_str:N #2 = true } }
            \c_false_bool { \exp_args:Ne #1 { \token_to_str:N #2 = false } }
          }
          {
            \msg_error:nneee { kernel } { bad-type }
              { \token_to_str:N #2 } { \token_to_meaning:N #2 } { bool }
          }
      }
  }
```

(*End of definition for* `\bool_show:N`*,* `\bool_log:N`*, and* `\__bool_show:NN`*. These functions are documented on page 68.*)

\l_tmpa_bool
\l_tmpb_bool
\g_tmpa_bool
\g_tmpb_bool

A few booleans just if you need them.

```
\bool_new:N \l_tmpa_bool
\bool_new:N \l_tmpb_bool
\bool_new:N \g_tmpa_bool
\bool_new:N \g_tmpb_bool
```

(*End of definition for* `\l_tmpa_bool` *and others. These variables are documented on page 68.*)

\bool_if_exist_p:N
\bool_if_exist_p:c
\bool_if_exist:N*TF*
\bool_if_exist:c*TF*

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \bool_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \bool_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End of definition for* `\bool_if_exist:NTF`*. This function is documented on page 68.*)

47.5 Boolean expressions

\bool_if_p:n
\bool_if:nTF

Evaluating the truth value of a list of predicates is done using an input syntax somewhat similar to the one found in other programming languages with `(` and `)` for grouping, `!` for logical "Not", `&&` for logical "And" and `||` for logical "Or". However, they perform eager evaluation. We shall use the terms Not, And, Or, Open and Close for these operations.

Any expression is terminated by a Close operation. Evaluation happens from left to right in the following manner using a GetNext function:

- If an Open is seen, start evaluating a new expression using the Eval function and call GetNext again.
- If a Not is seen, remove the `!` and call a GetNext function with the logic reversed.
- If none of the above, reinsert the token found (this is supposed to be a predicate function) in front of an Eval function, which evaluates it to the boolean value ⟨*true*⟩ or ⟨*false*⟩.

The Eval function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

⟨*true*⟩**And** Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

⟨*false*⟩**And** Current truth value is false, logical And seen, stop using the values of predicates within this sub-expression until the next Close. Then return ⟨*false*⟩.

⟨*true*⟩**Or** Current truth value is true, logical Or seen, stop using the values of predicates within this sub-expression until the nearest Close. Then return ⟨*true*⟩.

⟨*false*⟩**Or** Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

⟨*true*⟩**Close** Current truth value is true, Close seen, return ⟨*true*⟩.

⟨*false*⟩**Close** Current truth value is false, Close seen, return ⟨*false*⟩.

```
\prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
  {
    \if_predicate:w \bool_if_p:n {#1}
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\bool_if:nTF`. *This function is documented on page* *70*.)

\bool_if_p:n
__bool_if_p:n
__bool_if_p_aux:w

To speed up the case of a single predicate, `f`-expand and check whether the result is one token (possibly surrounded by spaces), which must be `\c_true_bool` or `\c_false_bool`. We use a version of `\tl_if_single:nTF` optimized for speed since we know that an empty `#1` is an error. The auxiliary `\__bool_if_p_aux:w` removes the trailing parenthesis and gets rid of any space, then returns `\c_true_bool` or `\c_false_bool` as appropriate. This extra work around is because in a `\bool_set:Nn`, the underlying `\chardef` turns the bool being set temporarily equal to \relax, thus assigning a boolean to itself would fail (gh/1055). For the general case, first issue a \group_align_safe_begin: as we are using && as syntax shorthand for the And operation and we need to hide it for TEX. This group is closed after __bool_get_next:NN returns \c_true_bool or \c_false_bool. That function requires the trailing parenthesis to know where the expression ends.

```
\cs_new:Npn \bool_if_p:n { \exp_args:Nf \__bool_if_p:n }
\cs_new:Npn \__bool_if_p:n #1
  {
    \tl_if_empty:oT { \use_none:nn #1 . } { \__bool_if_p_aux:w }
    \group_align_safe_begin:
    \exp_after:wN
    \group_align_safe_end:
    \exp:w \exp_end_continue_f:w % (
    \__bool_get_next:NN \use_i:nnnn #1 )
  }
\cs_new:Npn \__bool_if_p_aux:w #1 \use_i:nnnn #2#3
  { \bool_if:NTF #2 \c_true_bool \c_false_bool }
```

(*End of definition for \bool_if_p:n , __bool_if_p:n , and __bool_if_p_aux:w. This function is documented on page 70.*)

__bool_get_next:NN

The GetNext operation. Its first argument is \use_i:nnnn, \use_ii:nnnn, \use_iii:nnnn, or \use_iv:nnnn (we call these “states”). In the first state, this function eventually expand to the truth value \c_true_bool or \c_false_bool of the expression which follows until the next unmatched closing parenthesis. For instance “__bool_get_next:NN \use_i:nnnn \c_true_bool && \c_true_bool)” (including the closing parenthesis) expands to \c_true_bool. In the second state (after a !) the logic is reversed. We call these two states “normal” and the next two “skipping”. In the third state (after \c_true_bool||) it always returns \c_true_bool. In the fourth state (after \c_false_bool&&) it always returns \c_false_bool and also stops when encountering ||, not only parentheses. This code itself is a switch: if what follows is neither ! nor (, we assume it is a predicate.

```
\cs_new:Npn \__bool_get_next:NN #1#2
  {
    \use:c
      {
        __bool_
        \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
        :Nw
      }
      #1 #2
  }
```

(*End of definition for __bool_get_next:NN.*)

__bool_!:Nw

The Not operation reverses the logic: it discards the ! token and calls the GetNext operation with the appropriate first argument. Namely the first and second states are interchanged, but after \c_true_bool|| or \c_false_bool&& the ! is ignored.

```
\cs_new:cpn { __bool_!:Nw } #1#2
  {
    \exp_after:wN \__bool_get_next:NN
    #1 \use_ii:nnnn \use_i:nnnn \use_iii:nnnn \use_iv:nnnn
  }
```

(*End of definition for* `\__bool_!:Nw`.)

`\__bool_(:Nw`

The Open operation starts a sub-expression after discarding the open parenthesis. This is done by calling GetNext (which eventually discards the corresponding closing parenthesis), with a post-processing step which looks for And, Or or Close after the group.

```
\cs_new:cpn { __bool_(:Nw } #1#2
  {
    \exp_after:wN \__bool_choose:NNN \exp_after:wN #1
    \int_value:w \__bool_get_next:NN \use_i:nnnn
  }
```

(*End of definition for* `\__bool_(:Nw`.)

`\__bool_p:Nw`

If what follows GetNext is neither `!` nor `(`, evaluate the predicate using the primitive `\int_value:w`. The canonical `true` and `false` values have numerical values 1 and 0 respectively. Look for And, Or or Close afterwards.

```
\cs_new:cpn { __bool_p:Nw } #1
  { \exp_after:wN \__bool_choose:NNN \exp_after:wN #1 \int_value:w }
```

(*End of definition for* `\__bool_p:Nw`.)

`\__bool_choose:NNN`
`\__bool_)_0:`
`\__bool_)_1:`
`\__bool_)_2:`
`\__bool_&_0:`
`\__bool_&_1:`
`\__bool_&_2:`
`\__bool_|_0:`
`\__bool_|_1:`
`\__bool_|_2:`

The arguments are `#1`: a function such as `\use_i:nnnn`, `#2`: 0 or 1 encoding the current truth value, `#3`: the next operation, And, Or or Close. We distinguish three cases according to a combination of `#1` and `#2`. Case 2 is when `#1` is `\use_iii:nnnn` (state 3), namely after `\c_true_bool ||`. Case 1 is when `#1` is `\use_i:nnnn` and `#2` is `true` or when `#1` is `\use_ii:nnnn` and `#2` is `false`, for instance for `!\c_false_bool`. Case 0 includes the same with `true`/`false` interchanged and the case where `#1` is `\use_iv:nnnn` namely after `\c_false_bool &&`.

When seeing `)` the current subexpression is done, leave the appropriate boolean. When seeing `&` in case 0 go into state 4, equivalent to having seen `\c_false_bool &&`. In case 1, namely when the argument is `true` and we are in a normal state continue in the normal state 1. In case 2, namely when skipping alternatives in an Or, continue in the same state. When seeing `|` in case 0, continue in a normal state; in particular stop skipping for `\c_false_bool &&` because that binds more tightly than `||`. In the other two cases start skipping for `\c_true_bool ||`.

```
\cs_new:Npn \__bool_choose:NNN #1#2#3
  {
    \use:c
      {
        __bool_ \token_to_str:N #3 _
        #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: } 2 0 :
      }
  }
\cs_new:cpn { __bool_)_0: } { \c_false_bool }
\cs_new:cpn { __bool_)_1: } { \c_true_bool }
\cs_new:cpn { __bool_)_2: } { \c_true_bool }
\cs_new:cpn { __bool_&_0: } & { \__bool_get_next:NN \use_iv:nnnn }
\cs_new:cpn { __bool_&_1: } & { \__bool_get_next:NN \use_i:nnnn }
\cs_new:cpn { __bool_&_2: } & { \__bool_get_next:NN \use_iii:nnnn }
\cs_new:cpn { __bool_|_0: } | { \__bool_get_next:NN \use_i:nnnn }
\cs_new:cpn { __bool_|_1: } | { \__bool_get_next:NN \use_iii:nnnn }
\cs_new:cpn { __bool_|_2: } | { \__bool_get_next:NN \use_iii:nnnn }
```

(*End of definition for* __bool_choose:NNN *and others.*)

\bool_lazy_all_p:n
\bool_lazy_all:nTF
__bool_lazy_all:n

Go through the list of expressions, stopping whenever an expression is `false`. If the end is reached without finding any `false` expression, then the result is `true`.

```
\cs_new:Npn \bool_lazy_all_p:n #1
  { \__bool_lazy_all:n #1 \q__bool_recursion_tail \q__bool_recursion_stop }
\prg_new_conditional:Npnn \bool_lazy_all:n #1 { T , F , TF }
  {
    \if_predicate:w \bool_lazy_all_p:n {#1}
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \__bool_lazy_all:n #1
  {
    \__bool_if_recursion_tail_stop_do:nn {#1} { \c_true_bool }
    \bool_if:nF {#1}
      { \__bool_use_i_delimit_by_q_recursion_stop:nw { \c_false_bool } }
    \__bool_lazy_all:n
  }
```

(*End of definition for* \bool_lazy_all:nTF *and* __bool_lazy_all:n*. This function is documented on page 70.*)

\bool_lazy_and_p:nn
\bool_lazy_and:nnTF

Only evaluate the second expression if the first is `true`. Note that `#2` must be removed as an argument, not just by skipping to the `\else:` branch of the conditional since `#2` may contain unbalanced TeX conditionals.

```
\prg_new_conditional:Npnn \bool_lazy_and:nn #1#2 { p , T , F , TF }
  {
    \if_predicate:w
        \bool_if:nTF {#1} { \bool_if_p:n {#2} } { \c_false_bool }
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* \bool_lazy_and:nnTF*. This function is documented on page 70.*)

\bool_lazy_any_p:n
\bool_lazy_any:nTF
__bool_lazy_any:n

Go through the list of expressions, stopping whenever an expression is `true`. If the end is reached without finding any `true` expression, then the result is `false`.

```
\cs_new:Npn \bool_lazy_any_p:n #1
  { \__bool_lazy_any:n #1 \q__bool_recursion_tail \q__bool_recursion_stop }
\prg_new_conditional:Npnn \bool_lazy_any:n #1 { T , F , TF }
  {
    \if_predicate:w \bool_lazy_any_p:n {#1}
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \__bool_lazy_any:n #1
  {
```

```
    \__bool_if_recursion_tail_stop_do:nn {#1} { \c_false_bool }
    \bool_if:nT {#1}
      { \__bool_use_i_delimit_by_q_recursion_stop:nw { \c_true_bool } }
    \__bool_lazy_any:n
  }
```

(*End of definition for* `\bool_lazy_any:nTF` *and* `\__bool_lazy_any:n`*. This function is documented on page 70.*)

`\bool_lazy_or_p:nn`
`\bool_lazy_or:nnTF`

Only evaluate the second expression if the first is `false`.

```
\prg_new_conditional:Npnn \bool_lazy_or:nn #1#2 { p , T , F , TF }
  {
    \if_predicate:w
        \bool_if:nTF {#1} { \c_true_bool } { \bool_if_p:n {#2} }
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\bool_lazy_or:nnTF`*. This function is documented on page 70.*)

`\bool_not_p:n`

The Not variant just reverses the outcome of `\bool_if_p:n`. Can be optimized but this is nice and simple and according to the implementation plan. Not even particularly useful to have it when the infix notation is easier to use.

```
\cs_new:Npn \bool_not_p:n #1 { \bool_if_p:n { ! ( #1 ) } }
```

(*End of definition for* `\bool_not_p:n`*. This function is documented on page 70.*)

`\bool_xor_p:nn`
`\bool_xor:nnTF`

Exclusive or. If the boolean expressions have same truth value, return `false`, otherwise return `true`.

```
\prg_new_conditional:Npnn \bool_xor:nn #1#2 { p , T , F , TF }
  {
    \bool_if:nT {#1} \reverse_if:N
    \if_predicate:w \bool_if_p:n {#2}
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\bool_xor:nnTF`*. This function is documented on page 71.*)

47.6 Logical loops

`\bool_while_do:Nn`
`\bool_while_do:cn`
`\bool_until_do:Nn`
`\bool_until_do:cn`

A `while` loop where the boolean is tested before executing the statement. The "while" version executes the code as long as the boolean is true; the "until" version executes the code as long as the boolean is false.

```
\cs_new:Npn \bool_while_do:Nn #1#2
  { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
\cs_new:Npn \bool_until_do:Nn #1#2
  { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
\cs_generate_variant:Nn \bool_while_do:Nn { c }
\cs_generate_variant:Nn \bool_until_do:Nn { c }
```

(*End of definition for* `\bool_while_do:Nn` *and* `\bool_until_do:Nn`. *These functions are documented on page 71.*)

`\bool_do_while:Nn`
`\bool_do_while:cn`
`\bool_do_until:Nn`
`\bool_do_until:cn`

A `do-while` loop where the body is performed at least once and the boolean is tested after executing the body. Otherwise identical to the above functions.

```
\cs_new:Npn \bool_do_while:Nn #1#2
  { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
\cs_new:Npn \bool_do_until:Nn #1#2
  { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
\cs_generate_variant:Nn \bool_do_while:Nn { c }
\cs_generate_variant:Nn \bool_do_until:Nn { c }
```

(*End of definition for* `\bool_do_while:Nn` *and* `\bool_do_until:Nn`. *These functions are documented on page 71.*)

`\bool_while_do:nn`
`\bool_do_while:nn`
`\bool_until_do:nn`
`\bool_do_until:nn`

Loop functions with the test either before or after the first body expansion.

```
\cs_new:Npn \bool_while_do:nn #1#2
  {
    \bool_if:nT {#1}
      {
        #2
        \bool_while_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \bool_do_while:nn #1#2
  {
    #2
    \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
  }
\cs_new:Npn \bool_until_do:nn #1#2
  {
    \bool_if:nF {#1}
      {
        #2
        \bool_until_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \bool_do_until:nn #1#2
  {
    #2
    \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2}  }
  }
```

(*End of definition for* `\bool_while_do:nn` *and others. These functions are documented on page 72.*)

`\s__bool_mark`
`\s__bool_stop`

Internal scan marks.

```
\scan_new:N \s__bool_mark
\scan_new:N \s__bool_stop
```

(*End of definition for* `\s__bool_mark` *and* `\s__bool_stop`.)

`\bool_case:n`
`\bool_case:nTF`
`\__bool_case:NnTF`
`\__bool_case:w`
`a\__bool_case_end:nw`

For boolean cases the overall idea is the same as for `\str_case:nnTF` as described in l3str.

```
\cs_new:Npn \bool_case:nTF
  { \exp:w \__bool_case:nTF }
```

```
\cs_new:Npn \bool_case:nT #1#2
  { \exp:w \__bool_case:nTF {#1} {#2} { } }
\cs_new:Npn \bool_case:nF #1
  { \exp:w \__bool_case:nTF {#1} { } }
\cs_new:Npn \bool_case:n #1
  { \exp:w \__bool_case:nTF {#1} { } { } }
\cs_new:Npn \__bool_case:nTF #1#2#3
  {
    \__bool_case:w
    #1 \c_true_bool { } \s__bool_mark {#2} \s__bool_mark {#3} \s__bool_stop
  }
\cs_new:Npn \__bool_case:w #1#2
  {
    \bool_if:nTF {#1}
      { \__bool_case_end:nw {#2} }
      { \__bool_case:w }
  }
\cs_new:Npn \__bool_case_end:nw #1#2#3 \s__bool_mark #4#5 \s__bool_stop
  { \exp_end: #1 #4 }
```

(*End of definition for* `\bool_case:nTF` *and others. This function is documented on page 72.*)

47.7 Producing multiple copies

```
⟨@@=prg⟩
```

\prg_replicate:nn
__prg_replicate:N
__prg_replicate_first:N
__prg_replicate_
__prg_replicate_0:n
__prg_replicate_1:n
__prg_replicate_2:n
__prg_replicate_3:n
__prg_replicate_4:n
__prg_replicate_5:n
__prg_replicate_6:n
__prg_replicate_7:n
__prg_replicate_8:n
__prg_replicate_9:n
__prg_replicate_first_-:n
__prg_replicate_first_0:n
__prg_replicate_first_1:n
__prg_replicate_first_2:n
__prg_replicate_first_3:n
__prg_replicate_first_4:n
__prg_replicate_first_5:n
__prg_replicate_first_6:n
__prg_replicate_first_7:n
__prg_replicate_first_8:n
__prg_replicate_first_9:n

This function uses a cascading csname technique by David Kastrup (who else :-)

The idea is to make the input `25` result in first adding five, and then 20 copies of the code to be replicated. The technique uses cascading csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with `25` then the next function is the one that inserts two copies but it sees the ten copies handed down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. These functions also close the expansion of `\exp:w`, which ensures that `\prg_replicate:nn` only requires two steps of expansion.

This function has one flaw though: Since it constantly passes down ten copies of its previous argument it severely affects the main memory once you start demanding hundreds of thousands of copies. Now I don't think this is a real limitation for any ordinary use, and if necessary, it is possible to write `\prg_replicate:nn {1000} { \prg_replicate:nn {1000} {`⟨*code*⟩`} }`. An alternative approach is to create a string of `m`'s with `\exp:w` which can be done with just four macros but that method has its own problems since it can exhaust the string pool. Also, it is considerably slower than what we use here so the few extra csnames are well spent I would say.

```
\cs_new:Npn \prg_replicate:nn #1
  {
    \exp:w
      \exp_after:wN \__prg_replicate_first:N
      \int_value:w \int_eval:n {#1}
      \cs_end:
```

```
  }
\cs_new:Npn \__prg_replicate:N #1
  { \cs:w __prg_replicate_#1 :n \__prg_replicate:N }
\cs_new:Npn \__prg_replicate_first:N #1
  { \cs:w __prg_replicate_first_ #1 :n \__prg_replicate:N }
```

Then comes all the functions that do the hard work of inserting all the copies. The first function takes `:n` as a parameter.

```
\cs_new:Npn \__prg_replicate_ :n #1 { \cs_end: }
\cs_new:cpn { __prg_replicate_0:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} }
\cs_new:cpn { __prg_replicate_1:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1 }
\cs_new:cpn { __prg_replicate_2:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1 }
\cs_new:cpn { __prg_replicate_3:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1 }
\cs_new:cpn { __prg_replicate_4:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1 }
\cs_new:cpn { __prg_replicate_5:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_6:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_7:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_8:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_9:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1#1 }
```

Users shouldn't ask for something to be replicated once or even not at all but…

```
\cs_new:cpn { __prg_replicate_first_-:n } #1
  {
    \exp_end:
    \msg_expandable_error:nn { prg } { negative-replication }
  }
\cs_new:cpn { __prg_replicate_first_0:n } #1 { \exp_end: }
\cs_new:cpn { __prg_replicate_first_1:n } #1 { \exp_end: #1 }
\cs_new:cpn { __prg_replicate_first_2:n } #1 { \exp_end: #1#1 }
\cs_new:cpn { __prg_replicate_first_3:n } #1 { \exp_end: #1#1#1 }
\cs_new:cpn { __prg_replicate_first_4:n } #1 { \exp_end: #1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_5:n } #1 { \exp_end: #1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_6:n } #1 { \exp_end: #1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_7:n } #1 { \exp_end: #1#1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_8:n } #1 { \exp_end: #1#1#1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_9:n } #1
  { \exp_end: #1#1#1#1#1#1#1#1#1 }
```

(*End of definition for* `\prg_replicate:nn` *and others. This function is documented on page 72.*)

47.8 Detecting TeX's mode

`\mode_if_vertical_p:`
`\mode_if_vertical:TF`

For testing vertical mode. Strikes me here on the bus with David, that as long as we are just talking about returning true and false states, we can just use the primitive conditionals for this and gobbling the `\exp_end:` in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

```
\prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
  { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End of definition for* `\mode_if_vertical:TF`*. This function is documented on page 73.*)

`\mode_if_horizontal_p:`
`\mode_if_horizontal:TF`

For testing horizontal mode.

```
\prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
  { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End of definition for* `\mode_if_horizontal:TF`*. This function is documented on page 72.*)

`\mode_if_inner_p:`
`\mode_if_inner:TF`

For testing inner mode.

```
\prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
  { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End of definition for* `\mode_if_inner:TF`*. This function is documented on page 73.*)

`\mode_if_math_p:`
`\mode_if_math:TF`

For testing math mode. At the beginning of an alignment cell, this should be used only inside a non-expandable function.

```
\prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
  { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End of definition for* `\mode_if_math:TF`*. This function is documented on page 73.*)

47.9 Internal programming functions

`\group_align_safe_begin:`
`\group_align_safe_end:`

TeX's alignment structures present many problems. As Knuth says himself in *TeX: The Program*: "It's sort of a miracle whenever `\halign` or `\valign` work, [...]" One problem relates to commands that internally issue a `\cr` but also peek ahead for the next character for use in, say, an optional argument. If the next token happens to be a `&` with category code 4 we get some sort of weird error message because the underlying `\futurelet` stores the token at the end of the alignment template. This could be a `&`$_4$ giving a message like `! Misplaced \cr.` or even worse: it could be the `\endtemplate` token causing even more trouble! To solve this we have to open a special group so that TeX still thinks it's on safe ground but at the same time we don't want to introduce any brace group that may find its way to the output. The following functions help with this by using behaviour documented only in Appendix D of *The TeXbook*... In short evaluating `` `{ `` and `` `} `` as numbers will not change the counter TeX uses to keep track of its state in an alignment, whereas gobbling a brace using `\if_false:` will affect TeX's state without producing any real group. We place the `\if_false: { \fi:` part at that place so that the successive expansions of `\group_align_safe_begin/end:` are always brace balanced.

```
\group_begin:
\tex_catcode:D `\^^@ = 2 \exp_stop_f:
\cs_new:Npn \group_align_safe_begin:
  { \exp:w \if_false: { \fi: `^^@ \exp_stop_f: }
\tex_catcode:D `\^^@ = 1 \exp_stop_f:
\cs_new:Npn \group_align_safe_end:
  { \exp:w `^^@ \if_false: } \fi: \exp_stop_f: }
\group_end:
```

(*End of definition for* `\group_align_safe_begin:` *and* `\group_align_safe_end:`. *These functions are documented on page 74.*)

`\g__kernel_prg_map_int`

A nesting counter for mapping.

```
\int_new:N \g__kernel_prg_map_int
```

(*End of definition for* `\g__kernel_prg_map_int`.)

`\prg_break_point:Nn`
`\prg_map_break:Nn`

These are defined in l3basics, as they are needed "early". This is just a reminder that is the case!

(*End of definition for* `\prg_break_point:Nn` *and* `\prg_map_break:Nn`. *These functions are documented on page 73.*)

`\prg_break_point:`
`\prg_break:`
`\prg_break:n`

Also done in l3basics.

(*End of definition for* `\prg_break_point:`, `\prg_break:`, *and* `\prg_break:n`. *These functions are documented on page 74.*)

```
⟨/package⟩
```

Chapter 48

l3sys implementation

```
⟨@@=sys⟩
```

48.1 Kernel code

```
⟨*package⟩
⟨*tex⟩
```

\l__sys_tmp_tl

```
\tl_new:N \l__sys_tmp_tl
```

(*End of definition for* \l__sys_tmp_tl.)

48.1.1 Detecting the engine

__sys_const:nn

Set the T, F, TF, p forms of #1 to be constants equal to the result of evaluating the boolean expression #2.

```
\cs_new_protected:Npn \__sys_const:nn #1#2
  {
    \bool_if:nTF {#2}
      {
        \cs_new_eq:cN { #1 :T }  \use:n
        \cs_new_eq:cN { #1 :F }  \use_none:n
        \cs_new_eq:cN { #1 :TF } \use_i:nn
        \cs_new_eq:cN { #1 _p: } \c_true_bool
      }
      {
        \cs_new_eq:cN { #1 :T }  \use_none:n
        \cs_new_eq:cN { #1 :F }  \use:n
        \cs_new_eq:cN { #1 :TF } \use_ii:nn
        \cs_new_eq:cN { #1 _p: } \c_false_bool
      }
  }
```

(*End of definition for* __sys_const:nn.)

\sys_if_engine_luatex_p:
\sys_if_engine_luatex:TF
\sys_if_engine_pdftex_p:
\sys_if_engine_pdftex:TF
\sys_if_engine_ptex_p:
\sys_if_engine_ptex:TF
\sys_if_engine_uptex_p:
\sys_if_engine_uptex:TF
\sys_if_engine_xetex_p:
\sys_if_engine_xetex:TF
\c_sys_engine_str

Set up the engine tests on the basis exactly one test should be true. Mainly a case of looking for the appropriate marker primitive.

```
\str_const:Ne \c_sys_engine_str
```

```
  {
    \cs_if_exist:NT \tex_luatexversion:D { luatex }
    \cs_if_exist:NT \tex_pdftexversion:D { pdftex }
    \cs_if_exist:NT \tex_kanjiskip:D
      {
        \cs_if_exist:NTF \tex_enablecjktoken:D
          { uptex }
          { ptex }
      }
    \cs_if_exist:NT \tex_XeTeXversion:D { xetex }
  }
\tl_map_inline:nn { { luatex } { pdftex } { ptex } { uptex } { xetex } }
  {
    \__sys_const:nn { sys_if_engine_ #1 }
      { \str_if_eq_p:Vn \c_sys_engine_str {#1} }
  }
```

(*End of definition for* `\sys_if_engine_luatex:TF` *and others. These functions are documented on page* *76*.)

`\c_sys_engine_exec_str`
`\c_sys_engine_format_str`

Take the functions defined above, and set up the engine and format names. `\c_sys_engine_exec_str` differs from `\c_sys_engine_str` as it is the *actual* engine name, not a "filtered" version. It differs for `ptex` and `uptex`, which have a leading `e`, and for `luatex`, because LaTeX uses the LuaHBTeX engine.

`\c_sys_engine_format_str` is quite similar to `\c_sys_engine_str`, except that it differentiates `pdflatex` from `latex` (which is pdfTeX in DVI mode). This differentiation, however, is reliable only if the user doesn't change `\tex_pdfoutput:D` before loading this code.

```
\group_begin:
  \cs_set_eq:NN \lua_now:e    \tex_directlua:D
  \str_const:Ne \c_sys_engine_exec_str
    {
      \sys_if_engine_pdftex:T { pdf }
      \sys_if_engine_xetex:T  { xe  }
      \sys_if_engine_ptex:T   { ep  }
      \sys_if_engine_uptex:T  { eup }
      \sys_if_engine_luatex:T
        {
          lua \lua_now:e
            {
              if (pcall(require, 'luaharfbuzz')) then ~
                tex.print("hb") ~
              end
            }
        }
      tex
    }
\group_end:
\str_const:Ne \c_sys_engine_format_str
  {
    \cs_if_exist:NTF \fmtname
      {
        \bool_lazy_or:nnTF
          { \str_if_eq_p:Vn \fmtname { plain } }
```

```
          { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
          {
            \sys_if_engine_pdftex:T
              { \int_compare:nNnT { \tex_pdfoutput:D } = { 1 } { pdf } }
            \sys_if_engine_xetex:T  { xe }
            \sys_if_engine_ptex:T   { p  }
            \sys_if_engine_uptex:T  { up }
            \sys_if_engine_luatex:T
              {
                \int_compare:nNnT { \tex_pdfoutput:D } = { 0 } { dvi }
                lua
              }
            \str_if_eq:VnTF \fmtname { LaTeX2e }
              { latex }
              {
                \bool_lazy_and:nnT
                  { \sys_if_engine_pdftex_p: }
                  { \int_compare_p:nNn { \tex_pdfoutput:D } = { 0 } }
                    { e }
                tex
              }
          }
          { \fmtname }
      }
      { unknown }
  }
```

(*End of definition for* `\c_sys_engine_exec_str` *and* `\c_sys_engine_format_str`*. These variables are documented on page 76.*)

`\c_sys_engine_version_str` Various different engines, various different ways to extract the data!

```
\str_const:Ne \c_sys_engine_version_str
  {
    \str_case:on \c_sys_engine_str
      {
        { pdftex }
          {
            \int_div_truncate:nn { \tex_pdftexversion:D } { 100 }
            .
            \int_mod:nn { \tex_pdftexversion:D } { 100 }
            .
            \tex_pdftexrevision:D
          }
        { ptex }
          {
            \cs_if_exist:NT \tex_ptexversion:D
              {
                p
                \int_use:N  \tex_ptexversion:D
                .
                \int_use:N \tex_ptexminorversion:D
                \tex_ptexrevision:D
                -
                \int_use:N \tex_epTeXversion:D
```

```
                }
            }
          { luatex }
            {
              \int_div_truncate:nn { \tex_luatexversion:D } { 100 }
              .
              \int_mod:nn { \tex_luatexversion:D } { 100 }
              .
              \tex_luatexrevision:D
            }
          { uptex }
            {
              \cs_if_exist:NT \tex_ptexversion:D
                {
                  p
                  \int_use:N  \tex_ptexversion:D
                  .
                  \int_use:N \tex_ptexminorversion:D
                  \tex_ptexrevision:D
                  -
                  u
                  \int_use:N  \tex_uptexversion:D
                  \tex_uptexrevision:D
                  -
                  \int_use:N \tex_epTeXversion:D
                }
            }
          { xetex }
            {
              \int_use:N \tex_XeTeXversion:D
              \tex_XeTeXrevision:D
            }
        }
    }
```

(*End of definition for* `\c_sys_engine_version_str`. *This variable is documented on page 76.*)

48.1.2 Platform

\sys_if_platform_unix_p:
\sys_if_platform_unix:TF
\sys_if_platform_windows_p:
\sys_if_platform_windows:TF
\c_sys_platform_str

Setting these up requires the file module (file lookup), so is actually implemented there.

(*End of definition for* `\sys_if_platform_unix:TF`, `\sys_if_platform_windows:TF`, *and* `\c_sys_platform_str`. *These functions are documented on page 77.*)

48.1.3 Configurations

\sys_load_backend:n
__sys_load_backend_check:N
\c_sys_backend_str

Loading the backend code is pretty simply: check that the backend is valid, then load it up.

```
\cs_new_protected:Npn \sys_load_backend:n #1
  {
    \sys_finalise:
    \str_if_exist:NTF \c_sys_backend_str
      {
        \str_if_eq:VnF \c_sys_backend_str {#1}
```

```
          { \msg_error:nn { sys } { backend-set } }
      }
      {
        \tl_if_blank:nF {#1}
          { \tl_gset:Nn \g__sys_backend_tl {#1} }
        \__sys_load_backend_check:N \g__sys_backend_tl
        \str_const:Ne \c_sys_backend_str { \g__sys_backend_tl }
        \__kernel_sys_configuration_load:n
          { l3backend- \c_sys_backend_str }
      }
  }
\cs_new_protected:Npn \__sys_load_backend_check:N #1
  {
    \sys_if_engine_xetex:TF
      {
        \str_case:VnF #1
          {
            { dvisvgm }   { }
            { xdvipdfmx } { \tl_gset:Nn #1 { xetex } }
            { xetex }     { }
          }
          {
            \msg_error:nnee { sys } { wrong-backend }
              #1 { xetex }
            \tl_gset:Nn #1 { xetex }
          }
      }
      {
        \sys_if_output_pdf:TF
          {
            \str_if_eq:VnTF #1 { pdfmode }
              {
                \sys_if_engine_luatex:TF
                  { \tl_gset:Nn #1 { luatex } }
                  { \tl_gset:Nn #1 { pdftex } }
              }
              {
                \bool_lazy_or:nnF
                  { \str_if_eq_p:Vn #1 { luatex } }
                  { \str_if_eq_p:Vn #1 { pdftex } }
                  {
                    \msg_error:nnee { sys } { wrong-backend }
                      #1 { \sys_if_engine_luatex:TF { luatex } { pdftex } }
                    \sys_if_engine_luatex:TF
                      { \tl_gset:Nn #1 { luatex } }
                      { \tl_gset:Nn #1 { pdftex } }
                  }
              }
          }
          {
            \str_case:VnF #1
              {
                { dvipdfmx } { }
                { dvips }    { }
```

```
                    { dvisvgm }  { }
                  }
                  {
                    \msg_error:nnee { sys } { wrong-backend }
                      #1 { dvips }
                    \tl_gset:Nn #1 { dvips }
                  }
              }
          }
      }
```

(*End of definition for* `\sys_load_backend:n`, `\__sys_load_backend_check:N`, *and* `\c_sys_backend_str`. *These functions are documented on page 80.*)

`\sys_ensure_backend:` A simple wrapper.

```
\cs_new_protected:Npn \sys_ensure_backend:
  {
    \str_if_exist:NF \c_sys_backend_str
      { \sys_load_backend:n { } }
  }
```

(*End of definition for* `\sys_ensure_backend:`. *This function is documented on page 80.*)

`\g__sys_debug_bool`

```
\bool_new:N \g__sys_debug_bool
```

(*End of definition for* `\g__sys_debug_bool`.)

`\sys_load_debug:` The most complicated thing here is that we can only use `\__kernel_sys_configuration_load:n` in the preamble in LaTeX.

```
\cs_new_protected:Npn \sys_load_debug:
  {
    \bool_if:NF \g__sys_debug_bool
      { \__kernel_sys_configuration_load:n { l3debug } }
    \bool_gset_true:N \g__sys_debug_bool
  }
\cs_if_exist:NT \@expl@finalise@setup@@
  {
    \tl_gput_right:Nn \@expl@finalise@setup@@
      {
        \tl_gput_right:Nn \@kernel@after@begindocument
          {
            \cs_gset_protected:Npn \sys_load_debug:
              { \msg_error:nn { sys } { load-debug-in-preamble } }
          }
      }
  }
```

(*End of definition for* `\sys_load_debug:`. *This function is documented on page 80.*)

48.1.4 Access to the shell

\l__sys_internal_tl

```
\tl_new:N \l__sys_internal_tl
```

(*End of definition for* \l__sys_internal_tl.)

\c__sys_marker_tl

The same idea as the marker for rescanning token lists.

```
\tl_const:Ne \c__sys_marker_tl { : \token_to_str:N : }
```

(*End of definition for* \c__sys_marker_tl.)

\sys_get_shell:nnNTF
\sys_get_shell:nnN
__sys_get:nnN
__sys_get_do:Nw

Setting using a shell is at this level just a slightly specialised file operation, with an additional check for quotes, as these are not supported.

```
\cs_new_protected:Npn \sys_get_shell:nnN #1#2#3
  {
    \sys_get_shell:nnNF {#1} {#2} #3
      { \tl_set:Nn #3 { \q_no_value } }
  }
\prg_new_protected_conditional:Npnn \sys_get_shell:nnN #1#2#3 { T , F , TF }
  {
    \sys_if_shell:TF
      { \exp_args:No \__sys_get:nnN { \tl_to_str:n {#1} } {#2} #3 }
      { \prg_return_false: }
  }
\cs_new_protected:Npn \__sys_get:nnN #1#2#3
  {
    \tl_if_in:nnTF {#1} { " }
      {
        \msg_error:nne
          { kernel } { quote-in-shell } {#1}
        \prg_return_false:
      }
      {
        \group_begin:
          \if_false: { \fi:
          \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
          \exp_args:No \tex_everyeof:D { \c__sys_marker_tl }
          #2 \scan_stop:
          \exp_after:wN \__sys_get_do:Nw
          \exp_after:wN #3
          \exp_after:wN \prg_do_nothing:
            \tex_input:D | "#1" \scan_stop:
        \if_false: } \fi:
        \prg_return_true:
      }
  }
\exp_args:Nno \use:nn
  { \cs_new_protected:Npn \__sys_get_do:Nw #1#2 }
  { \c__sys_marker_tl }
  {
    \group_end:
    \tl_set:No #1 {#2}
  }
```

(*End of definition for* `\sys_get_shell:nnNTF` *and others. These functions are documented on page 78.*)

`\c__sys_shell_stream_int` This is not needed for LuaTeX: shell escape there isn't done using a TeX interface.

```
\sys_if_engine_luatex:F
  { \int_const:Nn \c__sys_shell_stream_int { 18 } }
```

(*End of definition for* `\c__sys_shell_stream_int`.)

`\sys_shell_now:n`
`\sys_shell_now:e`
`\sys_shell_now:x`
`\__sys_shell_now:e`

Execute commands through shell escape immediately.

For LuaTeX, we use a pseudo-primitive to do the actual work.

```
⟨/tex⟩
⟨*lua⟩
do
  local os_exec = os.execute

  local function shellescape(cmd)
    local status,msg = os_exec(cmd)
    if status == nil then
      write_nl("log","runsystem(" .. cmd .. ")...(" .. msg .. ")\n")
    elseif status == 0 then
      write_nl("log","runsystem(" .. cmd .. ")...executed\n")
    else
      write_nl("log","runsystem(" .. cmd .. ")...failed " .. (msg or "") .. "\n")
    end
  end
  luacmd("__sys_shell_now:e", function()
    shellescape(scan_string())
  end, "global", "protected")
⟨/lua⟩
```

```
⟨*tex⟩
\sys_if_engine_luatex:TF
  {
    \cs_new_protected:Npn \sys_shell_now:n #1
      { \__sys_shell_now:e { \exp_not:n {#1} } }
  }
  {
    \cs_new_protected:Npn \sys_shell_now:n #1
      { \iow_now:Nn \c__sys_shell_stream_int {#1} }
  }
\cs_generate_variant:Nn \sys_shell_now:n { e, x }
⟨/tex⟩
```

(*End of definition for* `\sys_shell_now:n` *and* `\__sys_shell_now:e`*. This function is documented on page 79.*)

`\sys_shell_shipout:n`
`\sys_shell_shipout:e`
`\sys_shell_shipout:x`
`\__sys_shell_shipout:e`

Execute commands through shell escape at shipout.

For LuaTeX, we use the same helper as above but delayed using a `late_lua` whatsit. Creating a `late_lua` whatsit works a bit different if we are running under ConTeXt.

```
⟨*lua⟩
  local new_latelua = nodes and nodes.nuts and nodes.nuts.pool and nodes.nuts.pool.latelua
    local whatsit_id = node.id'whatsit'
    local latelua_sub = node.subtype'late_lua'
    local node_new = node.direct.new
    local setfield = node.direct.setwhatsitfield or node.direct.setfield
```

```
    return function(f)
      local n = node_new(whatsit_id, latelua_sub)
      setfield(n, 'data', f)
      return n
    end
  end)()
  local node_write = node.direct.write

  luacmd("__sys_shell_shipout:e", function()
    local cmd = scan_string()
    node_write(new_latelua(function() shellescape(cmd) end))
  end, "global", "protected")
end
⟨/lua⟩
⟨*tex⟩
\sys_if_engine_luatex:TF
  {
    \cs_new_protected:Npn \sys_shell_shipout:n #1
    { \__sys_shell_shipout:e { \exp_not:n {#1} } }
  }
  {
    \cs_new_protected:Npn \sys_shell_shipout:n #1
      { \iow_shipout:Nn \c__sys_shell_stream_int {#1} }
  }
\cs_generate_variant:Nn \sys_shell_shipout:n { e , x }
```

(*End of definition for* `\sys_shell_shipout:n` *and* `\__sys_shell_shipout:e`. *This function is documented on page* 79.)

48.2 Dynamic (every job) code

`\__kernel_sys_everyjob:`
`\__sys_everyjob:n`
`\g__sys_everyjob_tl`

```
\cs_new_protected:Npn \__kernel_sys_everyjob:
  {
    \tl_use:N \g__sys_everyjob_tl
    \tl_gclear:N \g__sys_everyjob_tl
  }
\cs_new_protected:Npn \__sys_everyjob:n #1
  { \tl_gput_right:Nn \g__sys_everyjob_tl {#1} }
\tl_new:N \g__sys_everyjob_tl
```

(*End of definition for* `\__kernel_sys_everyjob:`, `\__sys_everyjob:n`, *and* `\g__sys_everyjob_tl`.)

48.2.1 The name of the job

`\c_sys_jobname_str` Inherited from the LaTeX3 name for the primitive. This *has* to be the primitive as it's set in `\everyjob`. If the user does

```
pdflatex \input some-file-name
```

then `\everyjob` is inserted *before* `\jobname` is changed form `texput`, and thus we would have the wrong result.

```
\__sys_everyjob:n
  { \cs_new_eq:NN \c_sys_jobname_str \tex_jobname:D }
```

(*End of definition for* `\c_sys_jobname_str`. *This variable is documented on page 75.*)

48.2.2 Time and date

`\c_sys_minute_int` `\c_sys_hour_int` `\c_sys_day_int` `\c_sys_month_int` `\c_sys_year_int`

Copies of the information provided by TEX. There is a lot of defensive code in package mode: someone may have moved the primitives, and they can only be recovered if we have `\primitive` and it is working correctly. For IniTEX of course that is all redundant but does no harm.

```
\__sys_everyjob:n
  {
    \group_begin:
      \cs_set:Npn \__sys_tmp:w #1
        {
          \str_if_eq:eeTF { \cs_meaning:N #1 } { \token_to_str:N #1 }
            { #1 }
            {
              \cs_if_exist:NTF \tex_primitive:D
                {
                  \bool_lazy_and:nnTF
                    { \sys_if_engine_xetex_p: }
                    {
                      \int_compare_p:nNn
                        { \exp_after:wN \use_none:n \tex_XeTeXrevision:D }
                          < { 99999 }
                    }
                    { 0 }
                    { \tex_primitive:D #1 }
                }
                { 0 }
            }
        }
      \int_const:Nn \c_sys_minute_int
        { \int_mod:nn { \__sys_tmp:w \time } { 60 } }
      \int_const:Nn \c_sys_hour_int
        { \int_div_truncate:nn { \__sys_tmp:w \time } { 60 } }
      \int_const:Nn \c_sys_day_int   { \__sys_tmp:w \day }
      \int_const:Nn \c_sys_month_int { \__sys_tmp:w \month }
      \int_const:Nn \c_sys_year_int  { \__sys_tmp:w \year }
    \group_end:
  }
```

(*End of definition for* `\c_sys_minute_int` *and others. These variables are documented on page 75.*)

`\c_sys_timestamp_str`

A simple expansion: LuaTEX chokes if we use `\pdffeedback` here, hence the direct use of Lua. Notice that the function there is in the `pdf` library but isn't actually tied to PDF.

```
\__sys_everyjob:n
  {
    \str_const:Ne \c_sys_timestamp_str
      {
        \cs_if_exist:NTF \tex_directlua:D
          { \tex_directlua:D { tex.print(pdf.getcreationdate()) } }
          { \tex_creationdate:D }
      }
  }
```

(*End of definition for* \c_sys_timestamp_str. *This variable is documented on page 75.*)

48.2.3 Random numbers

\sys_rand_seed: Unpack the primitive.

```
\__sys_everyjob:n
  {
    \cs_new:Npn \sys_rand_seed: { \tex_the:D \tex_randomseed:D }
  }
```

(*End of definition for* \sys_rand_seed:. *This function is documented on page 77.*)

\sys_gset_rand_seed:n The primitive always assigns the seed globally.

```
\__sys_everyjob:n
  {
    \cs_new_protected:Npn \sys_gset_rand_seed:n #1
      { \tex_setrandomseed:D \int_eval:n {#1} \exp_stop_f: }
  }
```

(*End of definition for* \sys_gset_rand_seed:n. *This function is documented on page 78.*)

\sys_timer:
__sys_elapsedtime:
\sys_if_timer_exist_p:
\sys_if_timer_exist:TF

In LuaTEX, create a pseudo-primitve, otherwise try to locate the real primitive. The elapsed time will be available if this succeeds.

```
⟨/tex⟩
⟨*lua⟩
  local gettimeofday = os.gettimeofday
  local epoch = gettimeofday() - os.clock()
  local write = tex.write
  local tointeger = math.tointeger
  luacmd('__sys_elapsedtime:', function()
    write(tointeger((gettimeofday() - epoch)*65536 // 1))
  end, 'global')
⟨/lua⟩
⟨*tex⟩
\sys_if_engine_luatex:TF
  {
    \cs_new:Npn \sys_timer:
      { \__sys_elapsedtime: }
  }
  {
    \cs_if_exist:NTF \tex_elapsedtime:D
      {
        \cs_new:Npn \sys_timer:
          { \int_value:w \tex_elapsedtime:D }
      }
      {
        \cs_new:Npn \sys_timer:
          {
            \int_value:w
            \msg_expandable_error:nnn { kernel } { no-elapsed-time }
              { \sys_timer: }
            \c_zero_int
          }
      }
```

```
  }
\__sys_const:nn { sys_if_timer_exist }
  { \cs_if_exist_p:N \tex_elapsedtime:D || \cs_if_exist_p:N \__sys_elapsedtime: }
```

(*End of definition for* `\sys_timer:` , `\__sys_elapsedtime:` , *and* `\sys_if_timer_exist:TF`. *These functions are documented on page 76.*)

48.2.4 Access to the shell

`\c_sys_shell_escape_int`

Expose the engine's shell escape status to the user.

```
\__sys_everyjob:n
  {
    \int_const:Nn \c_sys_shell_escape_int
      {
        \sys_if_engine_luatex:TF
          {
            \tex_directlua:D
              { tex.sprint(status.shell_escape~or~os.execute()) }
          }
          { \tex_shellescape:D }
      }
  }
```

(*End of definition for* `\c_sys_shell_escape_int`. *This variable is documented on page 78.*)

`\sys_if_shell_p:`
`\sys_if_shell:TF`
`\sys_if_shell_unrestricted_p:`
`\sys_if_shell_unrestricted:TF`
`\sys_if_shell_restricted_p:`
`\sys_if_shell_restricted:TF`

Performs a check for whether shell escape is enabled. The first set of functions returns true if either of restricted or unrestricted shell escape is enabled, while the other two sets of functions return true in only one of these two cases.

```
\__sys_everyjob:n
  {
    \__sys_const:nn { sys_if_shell }
      { \int_compare_p:nNn \c_sys_shell_escape_int > 0 }
    \__sys_const:nn { sys_if_shell_unrestricted }
      { \int_compare_p:nNn \c_sys_shell_escape_int = 1 }
    \__sys_const:nn { sys_if_shell_restricted }
      { \int_compare_p:nNn \c_sys_shell_escape_int = 2 }
  }
```

(*End of definition for* `\sys_if_shell:TF` , `\sys_if_shell_unrestricted:TF` , *and* `\sys_if_shell_restricted:TF`. *These functions are documented on page 78.*)

48.3 System queries

`\sys_get_query:nN`
`\sys_get_query:nnN`
`\sys_get_query:nnnN`
`\__sys_get_query_auxi:nnnN`
`\__sys_get_query_auxi:neeN`
`\__sys_get_query_auxii:nnnN`
`\__sys_get_query_auxii:neeN`

Calling the query system is quite straight-forward: most of the effort is in making the read-back catcode-safe. We also want to trim off the trailing `^^M` from the last line.

```
\cs_new_protected:Npn \sys_get_query:nN #1#2
  { \sys_get_query:nnnN {#1} { } { } #2 }
\cs_new_protected:Npn \sys_get_query:nnN #1#2#3
  { \sys_get_query:nnnN {#1} { } {#2} #3 }
\cs_new_protected:Npn \sys_get_query:nnnN #1#2#3#4
  {
    \tl_clear:N #4
    \__sys_get_query_auxi:neeN {#1} {#2} {#3} #4
```

```
  }
\cs_new:Npn \__sys_get_query_auxi:nnnN #1#2#3#4
  {
    \__sys_get_query_auxii:neeN {#1}
      { \tl_if_blank:nF {#2} { \tl_to_str:n { ~ #2 } } }
      {
        \tl_if_blank:nF {#3}
          {
            \c_space_tl
            \sys_if_shell_restricted:F '
            \tl_to_str:n {#3}
            \sys_if_shell_restricted:F '
          }
      }
      #4
  }
\cs_generate_variant:Nn \__sys_get_query_auxi:nnnN { nee }
\cs_new_protected:Npn \__sys_get_query_auxii:nnnN #1#2#3#4
  {
    \sys_if_shell:T
      {
        \sys_get_shell:nnN
          { l3sys-query~#1 #2 #3 }
          {
            \int_step_inline:nnn { 0 } { `A - 1 }
              { \char_set_catcode_other:n {##1} }
            \int_step_inline:nnn { `Z + 1 } { `a - 1 }
              { \char_set_catcode_other:n {##1} }
            \int_step_inline:nnn { `z + 1 } { 127 }
              { \char_set_catcode_other:n {##1} }
            \char_set_catcode_active:n { `\  }
            \tex_endlinechar:D 13 \scan_stop:
          }
        \l__sys_tmp_tl
        \tl_if_empty:NF \l__sys_tmp_tl
          {
            \exp_after:wN \__sys_get_query:Nw \exp_after:wN #4
              \l__sys_tmp_tl \q_stop
          }
      }
  }
\cs_generate_variant:Nn \__sys_get_query_auxii:nnnN { nee }
\group_begin:
  \tex_lccode:D `\* = 13 \scan_stop:
  \tex_lowercase:D
    {
      \group_end:
      \cs_new_protected:Npn \__sys_get_query:Nw #1#2 * \q_stop
    }
    { \tl_set:Nn #1 {#2} }
```

(*End of definition for* `\sys_get_query:nN` *and others. These functions are documented on page* 79.)

`\sys_split_query:nN`
`\sys_split_query:nnN`
`\sys_split_query:nnnN`

A wrapper for convenience.

```
\cs_new_protected:Npn \sys_split_query:nN #1#2
  { \sys_split_query:nnnN {#1} { } { } #2 }
\cs_new_protected:Npn \sys_split_query:nnN #1#2#3
  { \sys_split_query:nnnN {#1} { } {#2} #3 }
\group_begin:
  \tex_lccode:D `\* = 13 \scan_stop:
  \tex_lowercase:D
    {
      \group_end:
      \cs_new_protected:Npn \sys_split_query:nnnN #1#2#3#4
        {
          \seq_clear:N #4
          \sys_get_query:nnnN {#1} {#2} {#3} \l__sys_tmp_tl
          \tl_if_empty:NF \l__sys_tmp_tl
            { \seq_set_split:NnV #4 * \l__sys_tmp_tl }
        }
    }
```

(*End of definition for* `\sys_split_query:nN`, `\sys_split_query:nnN`, *and* `\sys_split_query:nnnN`. *These functions are documented on page 80.*)

48.3.1 Held over from l3file

`\g_file_curr_name_str` See comments about `\c_sys_jobname_str`: here, as soon as there is file input/output, things get "tided up".

```
\__sys_everyjob:n
  { \cs_gset_eq:NN \g_file_curr_name_str \tex_jobname:D }
```

(*End of definition for* `\g_file_curr_name_str`. *This variable is documented on page 100.*)

48.4 Last-minute code

`\sys_finalise:`
`\__sys_finalise:n`
`\g__sys_finalise_tl`

A simple hook to finalise the system-dependent layer. This is forced by the backend loader, which is forced by the main loader, so we do not need to include that here.

```
\cs_new_protected:Npn \sys_finalise:
  {
    \__kernel_sys_everyjob:
    \tl_use:N \g__sys_finalise_tl
    \tl_gclear:N \g__sys_finalise_tl
  }
\cs_new_protected:Npn \__sys_finalise:n #1
  { \tl_gput_right:Nn \g__sys_finalise_tl {#1} }
\tl_new:N \g__sys_finalise_tl
```

(*End of definition for* `\sys_finalise:`, `\__sys_finalise:n`, *and* `\g__sys_finalise_tl`. *This function is documented on page 80.*)

48.4.1 Detecting the output

`\sys_if_output_dvi_p:`
`\sys_if_output_dvi:TF`
`\sys_if_output_pdf_p:`
`\sys_if_output_pdf:TF`
`\c_sys_output_str`

This is a simple enough concept: the two views here are complementary.

```
\__sys_finalise:n
  {
    \str_const:Ne \c_sys_output_str
```

```
      {
        \int_compare:nNnTF
          { \cs_if_exist_use:NF \tex_pdfoutput:D { 0 } } > { 0 }
          { pdf }
          { dvi }
      }
    \__sys_const:nn { sys_if_output_dvi }
      { \str_if_eq_p:Vn \c_sys_output_str { dvi } }
    \__sys_const:nn { sys_if_output_pdf }
      { \str_if_eq_p:Vn \c_sys_output_str { pdf } }
  }
```

(*End of definition for* `\sys_if_output_dvi:TF`*,* `\sys_if_output_pdf:TF`*, and* `\c_sys_output_str`*. These functions are documented on page* *77**.*)

48.4.2 Configurations

`\g__sys_backend_tl`

As the backend has to be checked and possibly adjusted, the approach here is to create a variable and use that in a one-shot to set a constant.

```
\tl_new:N \g__sys_backend_tl
\__sys_finalise:n
  {
    \__kernel_tl_gset:Nx \g__sys_backend_tl
      {
        \sys_if_engine_xetex:TF
          { xetex }
          {
            \sys_if_output_pdf:TF
              {
                \sys_if_engine_pdftex:TF
                  { pdftex }
                  { luatex }
              }
              { dvips }
          }
      }
  }
```

If there is a class option set, and recognised, we pick it up: these will over-ride anything set automatically but will themselves be over-written if there is a package option.

```
\__sys_finalise:n
  {
    \cs_if_exist:NT \@classoptionslist
      {
        \cs_if_eq:NNF \@classoptionslist \scan_stop:
          {
            \clist_map_inline:Nn \@classoptionslist
              {
                \str_case:nnT {#1}
                  {
                    { dvipdfmx }
                      { \tl_gset:Nn \g__sys_backend_tl { dvipdfmx } }
                    { dvips }
                      { \tl_gset:Nn \g__sys_backend_tl { dvips } }
```

```
                        { dvisvgm }
                          { \tl_gset:Nn \g__sys_backend_tl { dvisvgm } }
                        { pdftex }
                          { \tl_gset:Nn \g__sys_backend_tl { pdfmode } }
                        { xetex }
                          { \tl_gset:Nn \g__sys_backend_tl { xdvipdfmx } }
                      }
                      { \clist_remove_all:Nn \@unusedoptionlist {#1} }
                  }
              }
          }
      }
```

(*End of definition for* `\g__sys_backend_tl`.)

```
⟨/tex⟩
⟨/package⟩
```

Chapter 49

l3msg implementation

```
⟨*package⟩
⟨@@=msg⟩
```

\l__msg_internal_tl

A general scratch for the module.

```
\tl_new:N \l__msg_internal_tl
```

(*End of definition for* `\l__msg_internal_tl`.)

\l__msg_name_str
\l__msg_text_str

Used to save module info when creating messages.

```
\str_new:N \l__msg_name_str
\str_new:N \l__msg_text_str
```

(*End of definition for* `\l__msg_name_str` *and* `\l__msg_text_str`.)

49.1 Internal auxiliaries

\s__msg_mark
\s__msg_stop

Internal scan marks.

```
\scan_new:N \s__msg_mark
\scan_new:N \s__msg_stop
```

(*End of definition for* `\s__msg_mark` *and* `\s__msg_stop`.)

__msg_use_none_delimit_by_s_stop:w

Functions to gobble up to a scan mark.

```
\cs_new:Npn \__msg_use_none_delimit_by_s_stop:w #1 \s__msg_stop { }
```

(*End of definition for* `\__msg_use_none_delimit_by_s_stop:w`.)

49.2 Creating messages

Messages are created and used separately, so there two parts to the code here. First, a mechanism for creating message text. This is pretty simple, as there is not actually a lot to do.

\c__msg_text_prefix_tl
\c__msg_more_text_prefix_tl

Locations for the text of messages.

```
\tl_const:Nn \c__msg_text_prefix_tl      { msg~text~>~ }
\tl_const:Nn \c__msg_more_text_prefix_tl { msg~extra~text~>~ }
```

(*End of definition for* \c__msg_text_prefix_tl *and* \c__msg_more_text_prefix_tl*.*)

\msg_if_exist_p:nn
\msg_if_exist:nnTF

Test whether the control sequence containing the message text exists or not.

```
\prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
  {
    \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
      { \prg_return_true: } { \prg_return_false: }
  }
```

(*End of definition for* \msg_if_exist:nnTF*. This function is documented on page 82.*)

__msg_chk_if_free:nn

This auxiliary is similar to __kernel_chk_if_free_cs:N, and is used when defining messages with \msg_new:nnnn.

```
\cs_new_protected:Npn \__msg_chk_free:nn #1#2
  {
    \msg_if_exist:nnT {#1} {#2}
      {
        \msg_error:nnnn { msg } { already-defined }
          {#1} {#2}
      }
  }
```

(*End of definition for* __msg_chk_if_free:nn*.*)

\msg_new:nnnn
\msg_new:nnee
\msg_new:nnxx
\msg_new:nnn
\msg_new:nne
\msg_new:nnx
\msg_set:nnnn
\msg_set:nnn

Setting a message simply means saving the appropriate text into two functions. A sanity check first.

```
\cs_new_protected:Npn \msg_new:nnnn #1#2#3#4
  {
    \__msg_chk_free:nn {#1} {#2}
    \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
      ##1##2##3##4 {#3}
    \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
      ##1##2##3##4 {#4}
  }
\cs_generate_variant:Nn \msg_new:nnnn { nnee , nnxx }
\cs_new_protected:Npn \msg_new:nnn #1#2#3
  { \msg_new:nnnn {#1} {#2} {#3} { } }
\cs_generate_variant:Nn \msg_new:nnn { nne , nnx }
\cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
  {
    \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
      ##1##2##3##4 {#3}
    \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
      ##1##2##3##4 {#4}
  }
\cs_new_protected:Npn \msg_set:nnn #1#2#3
  { \msg_set:nnnn {#1} {#2} {#3} { } }
```

(*End of definition for* \msg_new:nnnn *and others. These functions are documented on page 82.*)

49.3 Messages: support functions and text

\c__msg_coding_error_text_tl
\c__msg_continue_text_tl
\c__msg_critical_text_tl
\c__msg_fatal_text_tl
\c__msg_help_text_tl
\c__msg_no_info_text_tl
\c__msg_on_line_text_tl
\c__msg_return_text_tl
\c__msg_trouble_text_tl

Simple pieces of text for messages.

```
\tl_const:Nn \c__msg_coding_error_text_tl
  {
    This~is~a~coding~error.
    \\ \\
  }
\tl_const:Nn \c__msg_continue_text_tl
  { Type~<return>~to~continue }
\tl_const:Nn \c__msg_critical_text_tl
  { Reading~the~current~file~'\g_file_curr_name_str'~will~stop. }
\tl_const:Nn \c__msg_fatal_text_tl
  { This~is~a~fatal~error:~LaTeX~will~abort. }
\tl_const:Nn \c__msg_help_text_tl
  { For~immediate~help~type~H~<return> }
\tl_const:Nn \c__msg_no_info_text_tl
  {
    LaTeX~does~not~know~anything~more~about~this~error,~sorry.
    \c__msg_return_text_tl
  }
\tl_const:Nn \c__msg_on_line_text_tl { on~line }
\tl_const:Nn \c__msg_return_text_tl
  {
    \\ \\
    Try~typing~<return>~to~proceed.
    \\
    If~that~doesn't~work,~type~X~<return>~to~quit.
  }
\tl_const:Nn \c__msg_trouble_text_tl
  {
    \\ \\
    More~errors~will~almost~certainly~follow: \\
    the~LaTeX~run~should~be~aborted.
  }
```

(*End of definition for* `\c__msg_coding_error_text_tl` *and others.*)

\msg_line_number:
\msg_line_context:

For writing the line number nicely. `\msg_line_context:` was set up earlier, so this is not `new`.

```
\cs_new:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
\cs_gset:Npn \msg_line_context:
  {
    \c__msg_on_line_text_tl
    \c_space_tl
    \msg_line_number:
  }
```

(*End of definition for* `\msg_line_number:` *and* `\msg_line_context:`. *These functions are documented on page 83.*)

49.4 Showing messages: low level mechanism

__msg_interrupt:Nnnn
__msg_no_more_text:nnnn

The low-level interruption macro is rather opaque, unfortunately. Depending on the availability of more information there is a choice of how to set up the further help. We feed the extra help text and the message itself to a wrapping auxiliary, in this order because we must first setup TeX's `\errhelp` register before issuing an `\errmessage`. To deal with the various cases of critical or fatal errors with and without help text, there is a bit of argument-passing to do.

```
\cs_new_protected:Npn \__msg_interrupt:NnnnN #1#2#3#4#5
  {
    \str_set:Ne \l__msg_text_str { #1 {#2} }
    \str_set:Ne \l__msg_name_str { \msg_module_name:n {#2} }
    \cs_if_eq:cNTF
      { \c__msg_more_text_prefix_tl #2 / #3 }
      \__msg_no_more_text:nnnn
      {
        \__msg_interrupt_wrap:nnn
          { \use:c { \c__msg_text_prefix_tl #2 / #3 } #4 }
          { \c__msg_continue_text_tl }
          {
            \c__msg_no_info_text_tl
            \tl_if_empty:NF #5
              { \\ \\ #5 }
          }
      }
      {
        \__msg_interrupt_wrap:nnn
          { \use:c { \c__msg_text_prefix_tl #2 / #3 } #4 }
          { \c__msg_help_text_tl }
          {
            \use:c { \c__msg_more_text_prefix_tl #2 / #3 } #4
            \tl_if_empty:NF #5
              { \\ \\ #5 }
          }
      }
  }
\cs_new:Npn \__msg_no_more_text:nnnn #1#2#3#4 { }
```

(*End of definition for* `\__msg_interrupt:Nnnn` *and* `\__msg_no_more_text:nnnn`*.*)

__msg_interrupt_wrap:nnn
__msg_interrupt_text:n
__msg_interrupt_more_text:n

First setup TeX's `\errhelp` register with the extra help `#1`, then build a nice-looking error message with `#2`. Everything is done using `e`-type expansion as the new line markers are different for the two type of text and need to be correctly set up. The auxiliary `\__msg_interrupt_more_text:n` receives its argument as a line-wrapped string, which is thus unaffected by expansion. We ave to split the main text into two parts as only the "message" itself is wrapped with a leader: the generic help is wrapped at full width. We also have to allow for the two characters used by `\errmessage` itself.

```
\cs_new_protected:Npn \__msg_interrupt_wrap:nnn #1#2#3
  {
    \iow_wrap:nnnN { \\ #3 } { } { } \__msg_interrupt_more_text:n
    \group_begin:
      \int_sub:Nn \l_iow_line_count_int { 2 }
      \iow_wrap:nenN { \l__msg_text_str : ~ #1 }
```

```
        {
          ( \l__msg_name_str )
          \prg_replicate:nn
            {
                \str_count:N \l__msg_text_str
              - \str_count:N \l__msg_name_str
              + 2
            }
            { ~ }
        }
        { } \__msg_interrupt_text:n
    \iow_wrap:nnnN { \l__msg_internal_tl \\ \\ #2 } { } { }
      \__msg_interrupt:n
  }
\cs_new_protected:Npn \__msg_interrupt_text:n #1
  {
    \group_end:
    \tl_set:Nn \l__msg_internal_tl {#1}
  }
\cs_new_protected:Npn \__msg_interrupt_more_text:n #1
  { \exp_args:Ne \tex_errhelp:D { #1 \iow_newline: } }
```

(*End of definition for* `\__msg_interrupt_wrap:nnn`*,* `\__msg_interrupt_text:n`*, and* `\__msg_interrupt_more_text:n`*.*)

`\__msg_interrupt:n` The business end of the process starts by producing some visual separation of the message from the main part of the log. The error message needs to be printed with everything made "invisible": TeX's own information involves the macro in which `\errmessage` is called, and the end of the argument of the `\errmessage`, including the closing brace. We use an active `!` to call the `\errmessage` primitive, and end its argument with `\use_none:n {`⟨*spaces*⟩`}` which fills the output with spaces. Two trailing closing braces are turned into spaces to hide them as well. The group in which we alter the definition of the active `!` is closed before producing the message: this ensures that tokens inserted by typing `I` in the command-line are inserted after the message is entirely cleaned up.

The `\__kernel_iow_with:Nnn` auxiliary, defined in l3file, expects an ⟨*integer variable*⟩, an integer ⟨*value*⟩, and some ⟨*code*⟩. It runs the ⟨*code*⟩ after ensuring that the ⟨*integer variable*⟩ takes the given ⟨*value*⟩, then restores the former value of the ⟨*integer variable*⟩ if needed. We use it to ensure that the `\newlinechar` is 10, as needed for `\iow_newline:` to work, and that `\errorcontextlines` is -1, to avoid showing irrelevant context. Note that restoring the former value of these integers requires inserting tokens after the `\errmessage`, which go in the way of tokens which could be inserted by the user. This is unavoidable.

```
\group_begin:
  \char_set_lccode:nn { 38 } { 32 } % &
  \char_set_lccode:nn { 46 } { 32 } % .
  \char_set_lccode:nn { 123 } { 32 } % {
  \char_set_lccode:nn { 125 } { 32 } % }
  \char_set_catcode_active:N \&
\tex_lowercase:D
  {
    \group_end:
    \cs_new_protected:Npn \__msg_interrupt:n #1
      {
```

```
        \iow_term:n { }
        \__kernel_iow_with:Nnn \tex_newlinechar:D { `\^^J }
          {
            \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
              {
                \group_begin:
                  \cs_set_protected:Npn &
                    {
                      \tex_errmessage:D
                        {
                          #1
                          \use_none:n
                            { ............................................ }
                        }
                    }
                  \exp_after:wN
                \group_end:
                &
              }
          }
      }
  }
```

(*End of definition for* `\__msg_interrupt:n`.)

49.5 Displaying messages

LaTeX is handling error messages and so the TeX ones are disabled.

```
\int_gset:Nn \tex_errorcontextlines:D { -1 }
```

`\msg_fatal_text:n`
`\msg_critical_text:n`
`\msg_error_text:n`
`\msg_warning_text:n`
`\msg_info_text:n`
`\__msg_text:nn`
`\__msg_text:n`

A function for issuing messages: both the text and order could in principle vary. The module name may be empty for kernel messages, hence the slightly contorted code path for a space.

```
\cs_new:Npn \msg_fatal_text:n #1
  {
    Fatal ~
    \msg_error_text:n {#1}
  }
\cs_new:Npn \msg_critical_text:n #1
  {
    Critical ~
    \msg_error_text:n {#1}
  }
\cs_new:Npn \msg_error_text:n #1
  { \__msg_text:nn {#1} { Error } }
\cs_new:Npn \msg_warning_text:n #1
  { \__msg_text:nn {#1} { Warning } }
\cs_new:Npn \msg_info_text:n #1
  { \__msg_text:nn {#1} { Info } }
\cs_new:Npn \__msg_text:nn #1#2
  {
    \exp_args:Nf \__msg_text:n { \msg_module_type:n {#1} }
    \exp_args:Nf \__msg_text:n { \msg_module_name:n {#1} }
```

```
    #2
  }
\cs_new:Npn \__msg_text:n #1
  {
    \tl_if_blank:nF {#1}
      { #1 ~ }
  }
```

(*End of definition for* `\msg_fatal_text:n` *and others. These functions are documented on page* 83.)

`\g_msg_module_name_prop`
`\g_msg_module_type_prop`

For storing public module information: the kernel data is set up in advance.

```
\prop_new:N \g_msg_module_name_prop
\prop_new:N \g_msg_module_type_prop
\prop_gput:Nnn \g_msg_module_type_prop { LaTeX } { }
```

(*End of definition for* `\g_msg_module_name_prop` *and* `\g_msg_module_type_prop`*. These variables are documented on page* 82.)

`\msg_module_type:n`

Contextual footer information, with the potential to give modules an alternative name.

```
\cs_new:Npn \msg_module_type:n #1
  {
    \prop_if_in:NnTF \g_msg_module_type_prop {#1}
      { \prop_item:Nn \g_msg_module_type_prop {#1} }
      { Package }
  }
```

(*End of definition for* `\msg_module_type:n`*. This function is documented on page* 82.)

`\msg_module_name:n`
`\msg_see_documentation_text:n`

Contextual footer information, with the potential to give modules an alternative name.

```
\cs_new:Npn \msg_module_name:n #1
  {
    \prop_if_in:NnTF \g_msg_module_name_prop {#1}
      { \prop_item:Nn \g_msg_module_name_prop {#1} }
      {#1}
  }
\cs_new:Npn \msg_see_documentation_text:n #1
  {
    See~the~ \msg_module_name:n {#1} ~
    documentation~for~further~information.
  }
```

(*End of definition for* `\msg_module_name:n` *and* `\msg_see_documentation_text:n`*. These functions are documented on page* 82.)

`\__msg_class_new:nn`

```
\group_begin:
  \cs_set_protected:Npn \__msg_class_new:nn #1#2
    {
      \prop_new:c { l__msg_redirect_ #1 _prop }
      \cs_new_protected:cpn { __msg_ #1 _code:nnnnnn }
          ##1##2##3##4##5##6 {#2}
      \cs_new_protected:cpn { msg_ #1 :nnnnnn } ##1##2##3##4##5##6
        {
          \use:e
            {
```

```
              \exp_not:n { \__msg_use:nnnnnnn {#1} {##1} {##2} }
                { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
                { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
            }
        }
      \cs_new_protected:cpe { msg_ #1 :nnnnn } ##1##2##3##4##5
        { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
      \cs_new_protected:cpe { msg_ #1 :nnnn } ##1##2##3##4
        { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
      \cs_new_protected:cpe { msg_ #1 :nnn } ##1##2##3
        { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
      \cs_new_protected:cpe { msg_ #1 :nn } ##1##2
        { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
      \cs_generate_variant:cn { msg_ #1 :nnn }
        { nnV , nne , nnx }
      \cs_generate_variant:cn { msg_ #1 :nnnn }
        { nnVV , nnVn , nnnV , nnne , nnnx , nnee , nnxx }
      \cs_generate_variant:cn { msg_ #1 :nnnnn }
        { nnnee , nnnxx , nneee , nnxxx }
      \cs_generate_variant:cn { msg_ #1 :nnnnnn } { nneeee , nnxxxx }
    }
```

(*End of definition for* `\__msg_class_new:nn`.)

\msg_fatal:nnnnnn
\msg_fatal:nneeee
\msg_fatal:nnxxxx
\msg_fatal:nnnnn
\msg_fatal:nneee
\msg_fatal:nnxxx
\msg_fatal:nnnee
\msg_fatal:nnnxx
\msg_fatal:nnnn
\msg_fatal:nnVV
\msg_fatal:nnVn
\msg_fatal:nnnV
\msg_fatal:nnee
\msg_fatal:nnxx
\msg_fatal:nnnx
\msg_fatal:nnne
\msg_fatal:nnn
\msg_fatal:nnV
\msg_fatal:nne
\msg_fatal:nnx
\msg_fatal:nn
__msg_fatal_exit:

For fatal errors, after the error message TEX bails out. We force a bail out rather than using `\end` as this means it does not matter if we are in a context where normally the run cannot end.

```
    \__msg_class_new:nn { fatal }
      {
        \__msg_interrupt:NnnnN
          \msg_fatal_text:n {#1} {#2}
          { {#3} {#4} {#5} {#6} }
          \c__msg_fatal_text_tl
        \__msg_fatal_exit:
      }
    \cs_new_protected:Npn \__msg_fatal_exit:
      {
        \tex_batchmode:D
        \tex_read:D -1 to \l__msg_internal_tl
      }
```

(*End of definition for* `\msg_fatal:nnnnnn` *and others. These functions are documented on page 85.*)

\msg_critical:nnnnnn
\msg_critical:nneeee
\msg_critical:nnxxxx
\msg_critical:nnnnn
\msg_critical:nneee
\msg_critical:nnxxx
\msg_critical:nnnee
\msg_critical:nnnxx
\msg_critical:nnnn
\msg_critical:nnVV
\msg_critical:nnVn
\msg_critical:nnnV
\msg_critical:nnee
\msg_critical:nnxx
\msg_critical:nnnx
\msg_critical:nnne
\msg_critical:nnn
\msg_critical:nnV
\msg_critical:nne
\msg_critical:nnx
\msg_critical:nn

Not quite so bad: just end the current file.

```
    \__msg_class_new:nn { critical }
      {
        \__msg_interrupt:NnnnN
          \msg_critical_text:n {#1} {#2}
          { {#3} {#4} {#5} {#6} }
          \c__msg_critical_text_tl
        \tex_endinput:D
      }
```

(*End of definition for* `\msg_critical:nnnnnn` *and others. These functions are documented on page 85.*)

\msg_error:nnnnnn
\msg_error:nneeee
\msg_error:nnxxxx
\msg_error:nnnnn
\msg_error:nneee
\msg_error:nnxxx
\msg_error:nnnee
\msg_error:nnnxx
\msg_error:nnnn
\msg_error:nnVV
\msg_error:nnVn
\msg_error:nnnV
\msg_error:nnee
\msg_error:nnxx
\msg_error:nnnx
\msg_error:nnne
\msg_error:nnn
\msg_error:nnV
\msg_error:nne
\msg_error:nnx
\msg_error:nn

For an error, the interrupt routine is called. We check if there is a "more text" by comparing that control sequence with a permanently empty text. We have to undefine the bootstrap versions here.

```
\cs_undefine:N \msg_error:nnee
\cs_undefine:N \msg_error:nne
\cs_undefine:N \msg_error:nn
\__msg_class_new:nn { error }
  {
    \__msg_interrupt:NnnnN
      \msg_error_text:n {#1} {#2}
      { {#3} {#4} {#5} {#6} }
      \c_empty_tl
  }
```

(*End of definition for* `\msg_error:nnnnnn` *and others. These functions are documented on page* 85.)

__msg_info_aux:NNnnnnnn
\msg_warning:nnnnnn
\msg_warning:nneeee
\msg_warning:nnxxxx
\msg_warning:nnnnn
\msg_warning:nneee
\msg_warning:nnxxx
\msg_warning:nnnee
\msg_warning:nnnxx
\msg_warning:nnnn
\msg_warning:nnVV
\msg_warning:nnVn
\msg_warning:nnnV
\msg_warning:nnee
\msg_warning:nnxx
\msg_warning:nnnx
\msg_warning:nnne
\msg_warning:nnn
\msg_warning:nnV
\msg_warning:nne
\msg_warning:nnx
\msg_warning:nn
\msg_note:nnnnnn
\msg_note:nneeee
\msg_note:nnxxxx
\msg_note:nnnnn
\msg_note:nneee
\msg_note:nnxxx
\msg_note:nnnee
\msg_note:nnnxx
\msg_note:nnnn
\msg_note:nnVV
\msg_note:nnVn
\msg_note:nnnV
\msg_note:nnee
\msg_note:nnxx
\msg_note:nnnx
\msg_note:nnne
\msg_note:nnn
\msg_note:nnV
\msg_note:nne
\msg_note:nnx
\msg_note:nn
\msg_info:nnnnnn
\msg_info:nneeee

Warnings and information messages have no decoration. Warnings are printed to the terminal while information can either go to the log or both log and terminal.

```
\cs_new_protected:Npn \__msg_info_aux:NNnnnnnn #1#2#3#4#5#6#7#8
  {
    \str_set:Ne \l__msg_text_str { #2 {#3} }
    \str_set:Ne \l__msg_name_str { \msg_module_name:n {#3} }
    #1 { }
    \iow_wrap:nenN
      {
        \l__msg_text_str : ~
        \use:c { \c__msg_text_prefix_tl #3 / #4 } {#5} {#6} {#7} {#8}
      }
      {
        ( \l__msg_name_str )
        \prg_replicate:nn
          {
              \str_count:N \l__msg_text_str
            - \str_count:N \l__msg_name_str
          }
          { ~ }
      }
      { } #1
    #1 { }
  }
\__msg_class_new:nn { warning }
  {
    \__msg_info_aux:NNnnnnnn \iow_term:n \msg_warning_text:n
      {#1} {#2} {#3} {#4} {#5} {#6}
  }
\__msg_class_new:nn { note }
  {
    \__msg_info_aux:NNnnnnnn \iow_term:n \msg_info_text:n
      {#1} {#2} {#3} {#4} {#5} {#6}
  }
\__msg_class_new:nn { info }
  {
    \__msg_info_aux:NNnnnnnn \iow_log:n \msg_info_text:n
      {#1} {#2} {#3} {#4} {#5} {#6}
```

```
    }
```

(*End of definition for* `\__msg_info_aux:NNnnnnnn` *and others. These functions are documented on page 86.*)

\msg_term:nnnnnn
\msg_term:nneeee
\msg_term:nnxxxx
\msg_term:nnnnn
\msg_term:nneee
\msg_term:nnxxx
\msg_term:nnnee
\msg_term:nnnxx
\msg_term:nnnn
\msg_term:nnVV
\msg_term:nnVn
\msg_term:nnnV
\msg_term:nnee
\msg_term:nnxx
\msg_term:nnnx
\msg_term:nnne
\msg_term:nnn
\msg_term:nnV
\msg_term:nne
\msg_term:nnx
\msg_term:nn

"Log" data is very similar to information, but with no extras added. "Term" is used for communicating with the user through the terminal, like diagnostic messages, and debugging. This is similar to "log" messages, but uses the terminal output.

```
\__msg_class_new:nn { log }
  {
    \iow_wrap:nnnN
      { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
      { } { } \iow_log:n
  }
\__msg_class_new:nn { term }
  {
    \iow_wrap:nnnN
      { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
      { } { } \iow_term:n
  }
```

(*End of definition for* `\msg_term:nnnnnn` *and others. These functions are documented on page 87.*)

\msg_none:nnnnnn
\msg_none:nneeee
\msg_none:nnxxxx
\msg_none:nnnnn
\msg_none:nneee
\msg_none:nnxxx
\msg_none:nnnee
\msg_none:nnnxx
\msg_none:nnnn
\msg_none:nnVV
\msg_none:nnVn
\msg_none:nnnV
\msg_none:nnee
\msg_none:nnxx
\msg_none:nnnx
\msg_none:nnne
\msg_none:nnn
\msg_none:nnV
\msg_none:nne
\msg_none:nnx
\msg_none:nn

The `none` message type is needed so that input can be gobbled.

```
\__msg_class_new:nn { none } { }
```

(*End of definition for* `\msg_none:nnnnnn` *and others. These functions are documented on page 87.*)

\msg_show:nnnnnn
\msg_show:nneeee
\msg_show:nnxxxx
\msg_show:nnnnn
\msg_show:nneee
\msg_show:nnxxx
\msg_show:nnnee
\msg_show:nnnxx
\msg_show:nnnn
\msg_show:nnVV
\msg_show:nnVn
\msg_show:nnnV
\msg_show:nnee
\msg_show:nnxx
\msg_show:nnnx
\msg_show:nnne
\msg_show:nnn
\msg_show:nnV
\msg_show:nne
\msg_show:nnx
\msg_show:nn
__msg_show:n
__msg_show:w
__msg_show_dot:w
__msg_show:nn

The `show` message type is used for `\seq_show:N` and similar complicated data structures. Wrap the given text with a trailing dot (important later) then pass it to `\__msg_show:n`. If there is `\\>~` (or if the whole thing starts with `>~`) we split there, print the first part and show the second part using `\showtokens` (the `\exp_after:wN` ensure a nice display). Note that this primitive adds a leading `>~` and trailing dot. That is why we included a trailing dot before wrapping and removed it afterwards. If there is no `\\>~` do the same but with an empty second part which adds a spurious but inevitable `>~`.

```
\__msg_class_new:nn { show }
  {
    \iow_wrap:nnnN
      { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
      { } { } \__msg_show:n
  }
\cs_new_protected:Npn \__msg_show:n #1
  {
    \tl_if_in:nnTF { ^^J #1 } { ^^J > ~ }
      {
        \tl_if_in:nnTF { #1 \s__msg_mark } { . \s__msg_mark }
          { \__msg_show_dot:w } { \__msg_show:w }
        ^^J #1 \s__msg_stop
      }
      { \__msg_show:nn { ? #1 } { } }
  }
\cs_new:Npn \__msg_show_dot:w #1 ^^J > ~ #2 . \s__msg_stop
  { \__msg_show:nn {#1} {#2} }
\cs_new:Npn \__msg_show:w #1 ^^J > ~ #2 \s__msg_stop
```

```
    { \__msg_show:nn {#1} {#2} }
  \cs_new_protected:Npn \__msg_show:nn #1#2
    {
      \tl_if_empty:nF {#1}
        { \exp_args:No \iow_term:n { \use_none:n #1 } }
      \tl_set:Nn \l__msg_internal_tl {#2}
      \__kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
        {
          \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
            {
              \tex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
                { \exp_after:wN \l__msg_internal_tl }
            }
        }
    }
```

(*End of definition for* `\msg_show:nnnnnn` *and others. These functions are documented on page 88.*)

End the group to eliminate `\__msg_class_new:nn`.

```
\group_end:
```

`\msg_show_item:n`
`\msg_show_item_unbraced:n`
`\msg_show_item:nn`
`\msg_show_item_unbraced:nn`

Each item in the variable is formatted using one of the following functions. We cannot use `\\` and so on because these short-hands cannot be used inside the arguments of messages, only when defining the messages. We need to use `^^J` here directly as l3file is not yet loaded.

```
\cs_new:Npe \msg_show_item:n #1
  { ^^J > ~ \c_space_tl \exp_not:N \tl_to_str:n { {#1} } }
\cs_new:Npe \msg_show_item_unbraced:n #1
  { ^^J > ~ \c_space_tl \exp_not:N \tl_to_str:n {#1} }
\cs_new:Npe \msg_show_item:nn #1#2
  {
    ^^J > \use:nn { ~ } { ~ }
    \exp_not:N \tl_to_str:n { {#1} }
    \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
    \exp_not:N \tl_to_str:n { {#2} }
  }
\cs_new:Npe \msg_show_item_unbraced:nn #1#2
  {
    ^^J > \use:nn { ~ } { ~ }
    \exp_not:N \tl_to_str:n {#1}
    \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
    \exp_not:N \tl_to_str:n {#2}
  }
```

(*End of definition for* `\msg_show_item:n` *and others. These functions are documented on page 88.*)

`\__msg_class_chk_exist:nT`

Checking that a message class exists. We build this from `\cs_if_free:cTF` rather than `\cs_if_exist:cTF` because that avoids reading the second argument earlier than necessary.

```
\cs_new:Npn \__msg_class_chk_exist:nT #1
  {
    \cs_if_free:cTF { __msg_ #1 _code:nnnnnn }
      { \msg_error:nnn { msg } { class-unknown } {#1} }
  }
```

(*End of definition for* `\__msg_class_chk_exist:nT`.)

`\l__msg_class_tl`
`\l__msg_current_class_tl`

Support variables needed for the redirection system.

```
\tl_new:N \l__msg_class_tl
\tl_new:N \l__msg_current_class_tl
```

(*End of definition for* `\l__msg_class_tl` *and* `\l__msg_current_class_tl`.)

`\l__msg_redirect_prop`

For redirection of individually-named messages

```
\prop_new:N \l__msg_redirect_prop
```

(*End of definition for* `\l__msg_redirect_prop`.)

`\l__msg_hierarchy_seq`

During redirection, split the message name into a sequence: `{/module/submodule}`, `{/module}`, and `{}`.

```
\seq_new:N \l__msg_hierarchy_seq
```

(*End of definition for* `\l__msg_hierarchy_seq`.)

`\l__msg_class_loop_seq`

Classes encountered when following redirections to check for loops.

```
\seq_new:N \l__msg_class_loop_seq
```

(*End of definition for* `\l__msg_class_loop_seq`.)

`\__msg_use:nnnnnnn`
`\__msg_use_redirect_name:n`
`\__msg_use_hierarchy:nwwN`
`\__msg_use_redirect_module:n`
`\__msg_use_code:`

Actually using a message is a multi-step process. First, some safety checks on the message and class requested. The code and arguments are then stored to avoid passing them around. The assignment to `\__msg_use_code:` is similar to `\tl_set:Nn`. The message is eventually produced with whatever `\l__msg_class_tl` is when `\__msg_use_code:` is called. Here is also a good place to suppress tracing output if the `trace` package is loaded since all (non-expandable) messages go through this auxiliary.

```
\cs_new_protected:Npn \__msg_use:nnnnnnn #1#2#3#4#5#6#7
  {
    \cs_if_exist_use:N \conditionally@traceoff
    \msg_if_exist:nnTF {#2} {#3}
      {
        \__msg_class_chk_exist:nT {#1}
          {
            \tl_set:Nn \l__msg_current_class_tl {#1}
            \cs_set_protected:Npe \__msg_use_code:
              {
                \exp_not:n
                  {
                    \use:c { __msg_ \l__msg_class_tl _code:nnnnnn }
                      {#2} {#3} {#4} {#5} {#6} {#7}
                  }
              }
            \__msg_use_redirect_name:n { #2 / #3 }
          }
      }
      { \msg_error:nnnn { msg } { unknown } {#2} {#3} }
    \cs_if_exist_use:N \conditionally@traceon
  }
\cs_new_protected:Npn \__msg_use_code: { }
```

The first check is for a individual message redirection. If this applies then no further redirection is attempted. Otherwise, split the message name into ⟨*module*⟩, ⟨*submodule*⟩ and ⟨*message*⟩ (with an arbitrary number of slashes), and store `{/module/submodule}`, `{/module}` and `{}` into `\l__msg_hierarchy_seq`. We then map through this sequence, applying the most specific redirection.

```
\cs_new_protected:Npn \__msg_use_redirect_name:n #1
  {
    \prop_get:NnNTF \l__msg_redirect_prop { / #1 } \l__msg_class_tl
      { \__msg_use_code: }
      {
        \seq_clear:N \l__msg_hierarchy_seq
        \__msg_use_hierarchy:nwwN { }
          #1 \s__msg_mark \__msg_use_hierarchy:nwwN
          /  \s__msg_mark \__msg_use_none_delimit_by_s_stop:w
          \s__msg_stop
        \__msg_use_redirect_module:n { }
      }
  }
\cs_new_protected:Npn \__msg_use_hierarchy:nwwN #1#2 / #3 \s__msg_mark #4
  {
    \seq_put_left:Nn \l__msg_hierarchy_seq {#1}
    #4 { #1 / #2 } #3 \s__msg_mark #4
  }
```

At this point, the items of `\l__msg_hierarchy_seq` are the various levels at which we should look for a redirection. Redirections which are less specific than the argument of `\__msg_use_redirect_module:n` are not attempted. This argument is empty for a class redirection, `/module` for a module redirection, *etc.* Loop through the sequence to find the most specific redirection, with module `##1`. The loop is interrupted after testing for a redirection for `##1` equal to the argument `#1` (least specific redirection allowed). When a redirection is found, break the mapping, then if the redirection targets the same class, output the code with that class, and otherwise set the target as the new current class, and search for further redirections. Those redirections should be at least as specific as `##1`.

```
\cs_new_protected:Npn \__msg_use_redirect_module:n #1
  {
    \seq_map_inline:Nn \l__msg_hierarchy_seq
      {
        \prop_get:cnNTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
          {##1} \l__msg_class_tl
          {
            \seq_map_break:n
              {
                \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
                  { \__msg_use_code: }
                  {
                    \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
                    \__msg_use_redirect_module:n {##1}
                  }
              }
          }
          {
            \str_if_eq:nnT {##1} {#1}
```

```
              {
                \tl_set_eq:NN \l__msg_class_tl \l__msg_current_class_tl
                \seq_map_break:n { \__msg_use_code: }
              }
          }
      }
  }
```

(*End of definition for* `\__msg_use:nnnnnnn` *and others.*)

`\msg_redirect_name:nnn`

Named message always use the given class even if that class is redirected further. An empty target class cancels any existing redirection for that message.

```
\cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
  {
    \tl_if_empty:nTF {#3}
      { \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 } }
      {
        \__msg_class_chk_exist:nT {#3}
          { \prop_put:Nnn \l__msg_redirect_prop { / #1 / #2 } {#3} }
      }
  }
```

(*End of definition for* `\msg_redirect_name:nnn`*. This function is documented on page 90.*)

`\msg_redirect_class:nn`
`\msg_redirect_module:nnn`
`\__msg_redirect:nnn`
`\__msg_redirect_loop_chk:nnn`
`\__msg_redirect_loop_list:n`

If the target class is empty, eliminate the corresponding redirection. Otherwise, add the redirection. We must then check for a loop: as an initialization, we start by storing the initial class in `\l__msg_current_class_tl`.

```
\cs_new_protected:Npn \msg_redirect_class:nn
  { \__msg_redirect:nnn { } }
\cs_new_protected:Npn \msg_redirect_module:nnn #1
  { \__msg_redirect:nnn { / #1 } }
\cs_new_protected:Npn \__msg_redirect:nnn #1#2#3
  {
    \__msg_class_chk_exist:nT {#2}
      {
        \tl_if_empty:nTF {#3}
          { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
          {
            \__msg_class_chk_exist:nT {#3}
              {
                \prop_put:cnn { l__msg_redirect_ #2 _prop } {#1} {#3}
                \tl_set:Nn \l__msg_current_class_tl {#2}
                \seq_clear:N \l__msg_class_loop_seq
                \__msg_redirect_loop_chk:nnn {#2} {#3} {#1}
              }
          }
      }
  }
```

Since multiple redirections can only happen with increasing specificity, a loop requires that all steps are of the same specificity. The new redirection can thus only create a loop with other redirections for the exact same module, `#1`, and not submodules. After some initialization above, follow redirections with `\l__msg_class_tl`, and keep track in `\l__msg_class_loop_seq` of the various classes encountered. A redirection from a class to itself, or the absence of redirection both mean that there is no loop. A redirection to the initial class marks a loop. To break it, we must decide which redirection to cancel. The user most likely wants the newly added redirection to hold with no further redirection. We thus remove the redirection starting from `#2`, target of the new redirection. Note that no message is emitted by any of the underlying functions: otherwise we may get an infinite loop because of a message from the message system itself.

```
\cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
  {
    \seq_put_right:Nn \l__msg_class_loop_seq {#1}
    \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_class_tl
      {
        \str_if_eq:VnF \l__msg_class_tl {#1}
          {
            \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl
              {
                \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
                \msg_warning:nneeee
                  { msg } { redirect-loop }
                  { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
                  { \seq_item:Nn \l__msg_class_loop_seq { 2 } }
                  {#3}
                  {
                    \seq_map_function:NN \l__msg_class_loop_seq
                      \__msg_redirect_loop_list:n
                    { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
                  }
              }
              { \__msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} }
          }
      }
  }
\cs_generate_variant:Nn \__msg_redirect_loop_chk:nnn { o }
\cs_new:Npn \__msg_redirect_loop_list:n #1 { {#1} ~ => ~ }
```

(*End of definition for* `\msg_redirect_class:nn` *and others. These functions are documented on page 90.*)

49.6 Kernel-specific functions

`\__kernel_msg_show_eval:Nn`
`\__kernel_msg_log_eval:Nn`
`\__msg_show_eval:nnN`

A short-hand used for `\int_show:n` and similar functions that passes to `\tl_show:n` the result of applying `#1` (a function such as `\int_eval:n`) to the expression `#2`. The use of `f`-expansion ensures that `#1` is expanded in the scope in which the show command is called, rather than in the group created by `\iow_wrap:nnnN`. This is only important for expressions involving the `\currentgrouplevel` or `\currentgrouptype`. On the other hand we want the expression to be converted to a string with the usual escape character, hence within the wrapping code.

```
\cs_new_protected:Npn \__kernel_msg_show_eval:Nn #1#2
  { \exp_args:Nf \__msg_show_eval:nnN { #1 {#2} } {#2} \tl_show:n }
\cs_new_protected:Npn \__kernel_msg_log_eval:Nn #1#2
  { \exp_args:Nf \__msg_show_eval:nnN { #1 {#2} } {#2} \tl_log:n }
\cs_new_protected:Npn \__msg_show_eval:nnN #1#2#3 { #3 { #2 = #1 } }
```

(*End of definition for* `\__kernel_msg_show_eval:Nn`*,* `\__kernel_msg_log_eval:Nn`*, and* `\__msg_show_eval:nnN`*.*)

These are all retained purely for older xparse support.

`\__kernel_msg_new:nnnn`
`\__kernel_msg_new:nnn`

```
\cs_new_protected:Npn \__kernel_msg_new:nnnn #1
  { \msg_new:nnnn { LaTeX / #1 } }
\cs_new_protected:Npn \__kernel_msg_new:nnn #1
  { \msg_new:nnn { LaTeX / #1 } }
```

(*End of definition for* `\__kernel_msg_new:nnnn` *and* `\__kernel_msg_new:nnn`*.*)

`\__kernel_msg_info:nnxx`
`\__kernel_msg_warning:nnx`
`\__kernel_msg_warning:nnxx`
`\__kernel_msg_error:nnx`
`\__kernel_msg_error:nnxx`
`\__kernel_msg_error:nnxxx`

```
\cs_new_protected:Npn \__kernel_msg_info:nnxx #1
  { \msg_info:nnee { LaTeX / #1 } }
\cs_new_protected:Npn \__kernel_msg_warning:nnx #1
  { \msg_warning:nne { LaTeX / #1 } }
\cs_new_protected:Npn \__kernel_msg_warning:nnxx #1
  { \msg_warning:nnee { LaTeX / #1 } }
\cs_new_protected:Npn \__kernel_msg_error:nnx #1
  { \msg_error:nne { LaTeX / #1 } }
\cs_new_protected:Npn \__kernel_msg_error:nnxx #1
  { \msg_error:nnee { LaTeX / #1 } }
\cs_new_protected:Npn \__kernel_msg_error:nnxxx #1
  { \msg_error:nneee { LaTeX / #1 } }
```

(*End of definition for* `\__kernel_msg_info:nnxx` *and others.*)

`\__kernel_msg_expandable_error:nnn`
`\__kernel_msg_expandable_error:nnf`
`\__kernel_msg_expandable_error:nnff`

```
\cs_new:Npn \__kernel_msg_expandable_error:nnn #1
  { \msg_expandable_error:nnn { LaTeX / #1 } }
\cs_new:Npn \__kernel_msg_expandable_error:nnf #1
  { \msg_expandable_error:nnf { LaTeX / #1 } }
\cs_new:Npn \__kernel_msg_expandable_error:nnff #1
  { \msg_expandable_error:nnff { LaTeX / #1 } }
```

(*End of definition for* `\__kernel_msg_expandable_error:nnn` *and* `\__kernel_msg_expandable_error:nnff`*.*)

49.7 Internal messages

Error messages needed to actually implement the message system itself.

```
\msg_new:nnnn { msg } { already-defined }
  { Message~'#2'~for~module~'#1'~already~defined. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~was~asked~to~define~a~new~message~called~'#2'\\
    by~the~module~'#1':~this~message~already~exists.
    \c__msg_return_text_tl
  }
\msg_new:nnnn { msg } { unknown }
  { Unknown~message~'#2'~for~module~'#1'. }
  {
    \c__msg_coding_error_text_tl
```

```
    LaTeX~was~asked~to~display~a~message~called~'#2'\\
    by~the~module~'#1':~this~message~does~not~exist.
    \c__msg_return_text_tl
  }
\msg_new:nnnn { msg } { class-unknown }
  { Unknown~message~class~'#1'. }
  {
    LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\\
    this~was~never~defined.
    \c__msg_return_text_tl
  }
\msg_new:nnnn { msg } { redirect-loop }
  {
    Message~redirection~loop~caused~by~ {#1} ~=>~ {#2}
    \tl_if_empty:nF {#3} { ~for~module~' \use_none:n #3 ' } .
  }
  {
    Adding~the~message~redirection~ {#1} ~=>~ {#2}
    \tl_if_empty:nF {#3} { ~for~the~module~' \use_none:n #3 ' } ~
    created~an~infinite~loop\\\\
    \iow_indent:n { #4 \\\\ }
  }
```

Messages for earlier kernel modules plus a few for l3keys which cover coding errors.

```
\msg_new:nnnn { kernel } { bad-number-of-arguments }
  { Function~'#1'~cannot~be~defined~with~#2~arguments. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~define~a~function~'#1'~with~
    #2~arguments.~
    TeX~allows~between~0~and~9~arguments~for~a~single~function.
  }
\msg_new:nnnn { kernel } { command-already-defined }
  { Control~sequence~#1~already~defined. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~create~a~new~control~sequence~'#1'~
    but~this~name~has~already~been~used~elsewhere. \\ \\
    The~current~meaning~is:\\
    \ \ #2
  }
\msg_new:nnnn { kernel } { command-not-defined }
  { Control~sequence~#1~undefined. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~use~a~control~sequence~'#1':\\
    this~has~not~been~defined~yet.
  }
\msg_new:nnnn { kernel } { empty-search-pattern }
  { Empty~search~pattern. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'#1':~that~
    would~lead~to~an~infinite~loop!
  }
```

```
\cs_if_exist:NF \tex_elapsedtime:D
  {
    \msg_new:nnnn { kernel } { no-elapsed-time }
      { No~clock~detected~for~#1. }
      { The~current~engine~provides~no~way~to~access~the~system~time. }
  }
\msg_new:nnnn { kernel } { non-base-function }
  { Function~'#1'~is~not~a~base~function }
  {
    \c__msg_coding_error_text_tl
    Functions~defined~through~\iow_char:N\\cs_new:Nn~must~have~
    a~signature~consisting~of~only~normal~arguments~'N'~and~'n'.~
    The~signature~'#2'~of~'#1'~contains~other~arguments~'#3'.~
    To~define~variants~use~\iow_char:N\\cs_generate_variant:Nn~
    and~to~define~other~functions~use~\iow_char:N\\cs_new:Npn.
  }
\msg_new:nnnn { kernel } { missing-colon }
  { Function~'#1'~contains~no~':'. }
  {
    \c__msg_coding_error_text_tl
    Code-level~functions~must~contain~':'~to~separate~the~
    argument~specification~from~the~function~name.~This~is~
    needed~when~defining~conditionals~or~variants,~or~when~building~a~
    parameter~text~from~the~number~of~arguments~of~the~function.
  }
\msg_new:nnnn { kernel } { overflow }
  { Integers~larger~than~2^{30}-1~cannot~be~stored~in~arrays. }
  {
    An~attempt~was~made~to~store~#3~
    \tl_if_empty:nF {#2} { at~position~#2~ } in~the~array~'#1'.~
    The~largest~allowed~value~#4~will~be~used~instead.
  }
\msg_new:nnnn { kernel } { out-of-bounds }
  { Access~to~an~entry~beyond~an~array's~bounds. }
  {
    An~attempt~was~made~to~access~or~store~data~at~position~#2~of~the~
    array~'#1',~but~this~array~has~entries~at~positions~from~1~to~#3.
  }
\msg_new:nnnn { kernel } { protected-predicate }
  { Predicate~'#1'~must~be~expandable. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~define~'#1'~as~a~protected~predicate.~
    Only~expandable~tests~can~have~a~predicate~version.
  }
\msg_new:nnn { kernel } { randint-backward-range }
  { Wrong~order~of~bounds~in~\iow_char:N\\int_rand:nn{#1}{#2}. }
\msg_new:nnnn { kernel } { conditional-form-unknown }
  { Conditional~form~'#1'~for~function~'#2'~unknown. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~define~the~conditional~form~'#1'~of~
    the~function~'#2',~but~only~'TF',~'T',~'F',~and~'p'~forms~exist.
  }
```

```
\msg_new:nnnn { kernel } { variant-too-long }
  { Variant~form~'#1'~longer~than~base~signature~of~'#2'. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
    with~a~signature~starting~with~'#1',~but~that~is~longer~than~
    the~signature~(part~after~the~colon)~of~'#2'.
  }
\msg_new:nnnn { kernel } { invalid-variant }
  { Variant~form~'#1'~invalid~for~base~form~'#2'. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
    with~a~signature~starting~with~'#1',~but~cannot~change~an~argument~
    from~type~'#3'~to~type~'#4'.
  }
\msg_new:nnnn { kernel } { invalid-exp-args }
  { Invalid~variant~specifier~'#1'~in~'#2'. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~create~an~\iow_char:N\\exp_args:N...~
    function~with~signature~'N#2'~but~'#1'~is~not~a~valid~argument~
    specifier.
  }
\msg_new:nnn { kernel } { deprecated-variant }
  {
    Variant~form~'#1'~deprecated~for~base~form~'#2'.~
    One~should~not~change~an~argument~from~type~'#3'~to~type~'#4'
    \str_case:nnF {#3}
      {
        { n } { :~use~a~'\token_if_eq_charcode:NNTF #4 c v V'~variant? }
        { N } { :~base~form~only~accepts~a~single~token~argument. }
        {#4} { :~base~form~is~already~a~variant. }
      } { . }
  }
\msg_new:nnn { char } { active }
  { Cannot~generate~active~chars. }
\msg_new:nnn { char } { invalid-catcode }
  { Invalid~catcode~for~char~generation. }
\msg_new:nnn { char } { null-space }
  { Cannot~generate~null~char~as~a~space. }
\msg_new:nnn { char } { out-of-range }
  { Charcode~requested~out~of~engine~range. }
\msg_new:nnn { dim } { zero-unit }
  { Zero~unit~in~conversion. }
\msg_new:nnnn { kernel } { quote-in-shell }
  { Quotes~in~shell~command~'#1'. }
  { Shell~commands~cannot~contain~quotes~("). }
\msg_new:nnnn { keys } { no-property }
  { No~property~given~in~definition~of~key~'#1'. }
  {
    \c__msg_coding_error_text_tl
    Inside~\keys_define:nn  each~key~name~
    needs~a~property:  \\ \\
```

```
    \iow_indent:n { #1 .<property> } \\ \\
    LaTeX~did~not~find~a~'.'~to~indicate~the~start~of~a~property.
  }
\msg_new:nnnn { keys } { property-boolean-values-only }
  { The~property~'#1'~accepts~boolean~values~only. }
  {
    \c__msg_coding_error_text_tl
    The~property~'#1'~only~accepts~the~values~'true'~and~'false'.
  }
\msg_new:nnnn { keys } { property-requires-value }
  { The~property~'#1'~requires~a~value. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~was~asked~to~set~property~'#1'~for~key~'#2'.\\
    No~value~was~given~for~the~property,~and~one~is~required.
  }
\msg_new:nnnn { keys } { property-unknown }
  { The~key~property~'#1'~is~unknown. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
    this~property~is~not~defined.
  }
\msg_new:nnnn { quark } { invalid-function }
  { Quark~test~function~'#1'~is~invalid. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~create~quark~test~function~'#1'~
    \tl_if_empty:nTF {#2}
      { but~that~name~ }
      { with~signature~'#2',~but~that~signature~ }
    is~not~valid.
  }
\__kernel_msg_new:nnn { quark } { invalid }
  { Invalid~quark~variable~'#1'. }
\msg_new:nnnn { scanmark } { already-defined }
  { Scan~mark~#1~already~defined. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~create~a~new~scan~mark~'#1'~
    but~this~name~has~already~been~used~for~a~scan~mark.
  }
\msg_new:nnnn { seq } { item-too-large }
  { Sequence~'#1'~does~not~have~an~item~#3 }
  {
    An~attempt~was~made~to~push~or~pop~the~item~at~position~#3~
    of~'#1',~but~this~
    \int_compare:nTF { #3 = 0 }
      { position~does~not~exist. }
      { sequence~only~has~#2~item \int_compare:nF { #2 = 1 } {s}. }
  }
\msg_new:nnnn { seq } { shuffle-too-large }
  { The~sequence~#1~is~too~long~to~be~shuffled~by~TeX. }
  {
```

```
    TeX~has~ \int_eval:n { \c_max_register_int + 1 } ~
    toks~registers:~this~only~allows~to~shuffle~up~to~
    \int_use:N \c_max_register_int \ items.~
    The~list~will~not~be~shuffled.
  }
\msg_new:nnnn { kernel } { variable-not-defined }
  { Variable~#1~undefined. }
  {
    \c__msg_coding_error_text_tl
    LaTeX~has~been~asked~to~show~a~variable~#1,~but~this~has~not~
    been~defined~yet.
  }
\msg_new:nnnn { kernel } { bad-type }
  { Variable~'#1'~is~not~a~valid~#3. }
  {
    \c__msg_coding_error_text_tl
    The~variable~'#1'~with~\tl_if_empty:nTF {#4} {meaning} {value}\\\\
    \iow_indent:n {#2}\\\\
    should~be~a~#3~variable,~but~
    \tl_if_empty:nTF {#4}
      { it~is~not \str_if_eq:nnF {#3} { bool } { ~a~short~macro } . }
      {
        it~does~not~have~the~correct~
        \str_if_eq:nnTF {#2} {#4}
          { category~codes. }
          { internal~structure:\\\\\iow_indent:n {#4} }
      }
  }
\msg_new:nnnn { prop } { bad-link }
  { Variable~'#1'~is~not~a~valid~(linked)~prop. }
  {
    \c__msg_coding_error_text_tl
    The~variable~'#1'~has~an~incorrect~internal~structure.~
    Its~internal~entry~'#2'~points~to~'#3',~whose~name~is~not~of~the~
    form~'#4~<key>'.
  }
\msg_new:nnnn { clist } { non-clist }
  { Variable~'#1'~is~not~a~valid~clist. }
  {
    \c__msg_coding_error_text_tl
    The~variable~'#1'~with~value\\\\
    \iow_indent:n {#2}\\\\
    should~be~a~clist~variable,~but~it~includes~empty~or~blank~items~
    without~braces.
  }
\msg_new:nnnn { prop } { misused }
  { A~property~list~was~misused. }
  {
    \c__msg_coding_error_text_tl
    A~property~list~variable~was~used~without~an~accessor~function.~
    It~
    \tl_if_empty:nTF {#1}
      { is~empty. }
      { contains~the~key-value~pairs \use_none:n #1 . }
```

```
  }
\msg_new:nnnn { prop } { inner-make }
  { '#1'~ cannot~ be~ used~ in~ a~ group. }
  {
    \c__msg_coding_error_text_tl
    The~ command~ '#1'~ was~ applied~ to~ the~ property~ list~
    variable~ '#2', but~ the~ storage~ type~ can~ only~ be~ changed~
    at~ the~ outermost~ group~ level.
  }
```

Some errors only appear in expandable settings, hence don't need a "more-text" argument.

```
\msg_new:nnn { kernel } { bad-exp-end-f }
  { Misused~\exp_end_continue_f:w or~:nw }
\msg_new:nnn { kernel } { bad-variable }
  { Erroneous~variable~#1 used! }
\msg_new:nnn { seq } { misused }
  { A~sequence~was~misused. }
\msg_new:nnn { prg } { negative-replication }
  { Negative~argument~for~\iow_char:N\\prg_replicate:nn. }
\msg_new:nnn { prop } { prop-keyval }
  { Missing~'='~in~'#1'~(in~'..._keyval:Nn') }
\msg_new:nnn { kernel } { unknown-comparison }
  { Relation~'#1'~not~among~=,<,>,==,!=,<=,>=. }
\msg_new:nnn { kernel } { zero-step }
  { Zero~step~size~for~function~#1. }
```

Messages used by the "show" functions.

```
\msg_new:nnn { clist } { show }
  {
    The~comma~list~ \tl_if_empty:nF {#1} { #1 ~ }
    \tl_if_empty:nTF {#2}
      { is~empty \\>~ . }
      { contains~the~items~(without~outer~braces): #2 . }
  }
\msg_new:nnn { intarray } { show }
  { The~integer~array~#1~contains~#2~items: \\ #3 . }
\msg_new:nnn { prop } { show }
  {
    The~ \str_if_eq:nnF {#3} { flat } { #3~ }
    property~list~#1~
    \tl_if_empty:nTF {#2}
      { is~empty \\>~ . }
      { contains~the~pairs~(without~outer~braces): #2 . }
  }
\msg_new:nnn { seq } { show }
  {
    The~sequence~#1~
    \tl_if_empty:nTF {#2}
      { is~empty \\>~ . }
      { contains~the~items~(without~outer~braces): #2 . }
  }
\msg_new:nnn { kernel } { show-streams }
  {
    \tl_if_empty:nTF {#2} { No~ } { The~following~ }
```

```
    \str_case:nn {#1}
      {
        { ior } { input ~ }
        { iow } { output ~ }
      }
    streams~are~
    \tl_if_empty:nTF {#2} { open } { in~use: #2 . }
  }
```

System layer messages

```
\msg_new:nnnn { sys } { backend-set }
  { Backend~configuration~already~set. }
  {
    Run-time~backend~selection~may~only~be~carried~out~once~during~a~run.~
    This~second~attempt~to~set~them~will~be~ignored.
  }
\msg_new:nnnn { sys } { load-debug-in-preamble }
  { Load~debug~support~in~the~preamble. }
  {
    Debugging~requires~support~loaded~in~the~preamble: \\
    Use~\sys_load_debug:~before~\begin{document}.
  }
\msg_new:nnnn { sys } { wrong-backend }
  { Backend~request~inconsistent~with~engine:~using~'#2'~backend. }
  {
    You~have~requested~backend~'#1',~but~this~is~not~suitable~for~use~with~the~
    active~engine.~LaTeX~will~use~the~'#2'~backend~instead.
  }
```

49.8 Expandable errors

__msg_expandable_error:nn

In expansion only context, we cannot use the normal means of reporting errors. Instead, we rely on a low-level TeX error caused by expanding a macro `\???` with parameter text "`?`" (this could be any token) which we used followed by something else (here, a space). This shows the context, which thanks to the odd-looking `\use:n` is

```
<argument> \???
                 ! mypkg Error: The error message.
```

In other words, TeX is processing the argument of `\use:n`, which is `\???` ⟨*space*⟩ `!` ⟨*error type*⟩ `:` ⟨*error message*⟩.

```
\cs_set_protected:Npn \__msg_tmp:w #1
  {
    \cs_new:Npn #1 ? { }
    \cs_new:Npn \__msg_expandable_error:nn ##1##2
      {
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__msg_use_none_delimit_by_s_stop:w
        \use:n { #1 ~ ! ~ ##2 : ~ ##1 } \s__msg_stop
      }
  }
\exp_args:Nc \__msg_tmp:w { ??? }
```

(*End of definition for* `\__msg_expandable_error:nn`.)

\msg_expandable_error:nnnnnn
\msg_expandable_error:nnffff
\msg_expandable_error:nnnnn
\msg_expandable_error:nnfff
\msg_expandable_error:nnnn
\msg_expandable_error:nnff
\msg_expandable_error:nnn
\msg_expandable_error:nnf
\msg_expandable_error:nn

The command built from the csname `\c__msg_text_prefix_tl #1 / #2` takes four arguments and builds the error text, which is fed to `\__msg_expandable_error:n` with appropriate expansion: just as for usual messages the arguments are first turned to strings, then the message is fully expanded. The module name also has to be determined.

```
\exp_args_generate:n { oooo }
\cs_new:Npn \msg_expandable_error:nnnnnn #1#2#3#4#5#6
  {
    \exp_args:Nee \__msg_expandable_error:nn
      {
        \exp_args:Nc \exp_args:Noooo
          { \c__msg_text_prefix_tl #1 / #2 }
          { \tl_to_str:n {#3} }
          { \tl_to_str:n {#4} }
          { \tl_to_str:n {#5} }
          { \tl_to_str:n {#6} }
      }
      { \msg_error_text:n {#1} }
  }
\cs_new:Npn \msg_expandable_error:nnnnn #1#2#3#4#5
  { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }
\cs_new:Npn \msg_expandable_error:nnnn #1#2#3#4
  { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} { } { } }
\cs_new:Npn \msg_expandable_error:nnn #1#2#3
  { \msg_expandable_error:nnnnnn {#1} {#2} {#3} { } { } { } }
\cs_new:Npn \msg_expandable_error:nn #1#2
  { \msg_expandable_error:nnnnnn {#1} {#2} { } { } { } { } }
\cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnffff }
\cs_generate_variant:Nn \msg_expandable_error:nnnnn  { nnfff }
\cs_generate_variant:Nn \msg_expandable_error:nnnn   { nnff }
\cs_generate_variant:Nn \msg_expandable_error:nnn    { nnf }
```

(*End of definition for* `\msg_expandable_error:nnnnnn` *and others. These functions are documented on page 89.*)

49.9 Message formatting

```
\prop_gput:Nnn \g_msg_module_name_prop { kernel } { LaTeX }
\prop_gput:Nnn \g_msg_module_type_prop { kernel } { }
\clist_map_inline:nn
  {
    char , clist , coffin , debug , deprecation , dim, msg ,
    quark , prg , prop , scanmark , seq , sys
  }
  {
    \prop_gput:Nnn \g_msg_module_name_prop {#1} { LaTeX }
    \prop_gput:Nnn \g_msg_module_type_prop {#1} { }
  }
\prop_gput:Nnn \g_msg_module_name_prop { LaTeX / cmd } { LaTeX }
\prop_gput:Nnn \g_msg_module_type_prop { LaTeX / cmd } { }
\prop_gput:Nnn \g_msg_module_name_prop { LaTeX / ltcmd } { LaTeX }
\prop_gput:Nnn \g_msg_module_type_prop { LaTeX / ltcmd } { }
⟨/package⟩
```

Chapter 50

l3file implementation

The following test files are used for this code: m3file001.

```
⟨*package⟩
```

50.1 Input operations

```
⟨@@=ior⟩
```

50.1.1 Variables and constants

\l__ior_internal_tl Used as a short-term scratch variable.

```
\tl_new:N  \l__ior_internal_tl
```

(*End of definition for* \l__ior_internal_tl*.*)

\c__ior_term_ior Reading from the terminal (with a prompt) is done using a positive but non-existent stream number. Unlike writing, there is no concept of reading from the log.

```
\int_const:Nn \c__ior_term_ior { 16 }
```

(*End of definition for* \c__ior_term_ior*.*)

\g__ior_streams_seq A list of the currently-available input streams to be used as a stack.

```
\seq_new:N \g__ior_streams_seq
```

(*End of definition for* \g__ior_streams_seq*.*)

\l__ior_stream_tl Used to recover the raw stream number from the stack.

```
\tl_new:N \l__ior_stream_tl
```

(*End of definition for* \l__ior_stream_tl*.*)

\g__ior_streams_prop The name of the file attached to each stream is tracked in a property list. To get the correct number of reserved streams in package mode the underlying mechanism needs to be queried. For LaTeX 2_ε and plain TeX this data is stored in `\count16`: with the `etex` package loaded we need to subtract 1 as the register holds the number of the next stream to use. In ConTeXt, we need to look at `\count38` but there is no subtraction: like the original plain TeX/LaTeX 2_ε mechanism it holds the value of the *last* stream allocated.

```
\prop_new:N \g__ior_streams_prop
```

```
\int_step_inline:nnn
  { 0 }
  {
    \cs_if_exist:NTF \contextversion
      { \tex_count:D 38 ~ }
      {
        \tex_count:D 16 ~ %
        \cs_if_exist:NT \loccount { - 1 }
      }
  }
  {
    \prop_gput:Nnn \g__ior_streams_prop {#1} { Reserved~by~format }
  }
```

(*End of definition for* `\g__ior_streams_prop`.)

50.1.2 Stream management

`\ior_new:N`
`\ior_new:c`

Reserving a new stream is done by defining the name as equal to using the terminal.

```
\cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c__ior_term_ior }
\cs_generate_variant:Nn \ior_new:N { c }
```

(*End of definition for* `\ior_new:N`. *This function is documented on page 92.*)

`\g_tmpa_ior`
`\g_tmpb_ior`

The usual scratch space.

```
\ior_new:N \g_tmpa_ior
\ior_new:N \g_tmpb_ior
```

(*End of definition for* `\g_tmpa_ior` *and* `\g_tmpb_ior`. *These variables are documented on page 100.*)

`\ior_open:Nn`
`\ior_open:cn`

Use the conditional version, with an error if the file is not found.

```
\cs_new_protected:Npn \ior_open:Nn #1#2
  { \ior_open:NnF #1 {#2} { \__kernel_file_missing:n {#2} } }
\cs_generate_variant:Nn \ior_open:Nn { c }
```

(*End of definition for* `\ior_open:Nn`. *This function is documented on page 92.*)

`\l__ior_file_name_tl`

Data storage.

```
\tl_new:N \l__ior_file_name_tl
```

(*End of definition for* `\l__ior_file_name_tl`.)

`\ior_open:NnTF`
`\ior_open:cnTF`

An auxiliary searches for the file in the TeX, LaTeX 2_ε and LaTeX3 paths. Then pass the file found to the lower-level function which deals with streams. The `full_name` is empty when the file is not found.

```
\prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
  {
    \file_get_full_name:nNTF {#2} \l__ior_file_name_tl
      {
        \__kernel_ior_open:No #1 \l__ior_file_name_tl
        \prg_return_true:
      }
      { \prg_return_false: }
  }
\prg_generate_conditional_variant:Nnn \ior_open:Nn { c } { T , F , TF }
```

(*End of definition for* `\ior_open:NnTF`. *This function is documented on page 92.*)

`\__ior_new:N` Streams are reserved using `\newread` before they can be managed by ior. To prevent ior from being affected by redefinitions of `\newread` (such as done by the third-party package morewrites), this macro is saved here under a private name. The complicated code ensures that `\__ior_new:N` is not `\outer` despite plain TeX's `\newread` being `\outer`. For ConTeXt, we have to deal with the fact that `\newread` works like our own: it actually checks before altering definition.

```
\exp_args:NNf \cs_new_protected:Npn \__ior_new:N
  { \exp_args:NNc \exp_after:wN \exp_stop_f: { newread } }
\cs_if_exist:NT \contextversion
  {
    \cs_new_eq:NN \__ior_new_aux:N \__ior_new:N
    \cs_gset_protected:Npn \__ior_new:N #1
      {
        \cs_undefine:N #1
        \__ior_new_aux:N #1
      }
  }
```

(*End of definition for* `\__ior_new:N`.)

`\__kernel_ior_open:Nn`
`\__kernel_ior_open:No`
`\__ior_open_stream:Nn`

The stream allocation itself uses the fact that there is a list of all of those available. Life gets more complex as it's important to keep things in sync. That is done using a two-part approach: any streams that have already been taken up by ior but are now free are tracked, so we first try those. If that fails, ask plain TeX or LaTeX 2ε for a new stream and use that number (after a bit of conversion).

```
\cs_new_protected:Npn \__kernel_ior_open:Nn #1#2
  {
    \ior_close:N #1
    \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
      { \__ior_open_stream:Nn #1 {#2} }
      {
        \__ior_new:N #1
        \__kernel_tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
        \__ior_open_stream:Nn #1 {#2}
      }
  }
\cs_generate_variant:Nn \__kernel_ior_open:Nn { No }
```

Here, we act defensively in case LuaTeX is in use with an extensionless file name.

```
\cs_new_protected:Npe \__ior_open_stream:Nn #1#2
  {
    \tex_global:D \tex_chardef:D #1 = \exp_not:N \l__ior_stream_tl \scan_stop:
    \prop_gput:NVn \exp_not:N \g__ior_streams_prop #1 {#2}
    \tex_openin:D #1
      \sys_if_engine_luatex:TF
        { {#2} }
        {  \exp_not:N \__kernel_file_name_quote:n {#2} \scan_stop: }
  }
```

(*End of definition for* `\__kernel_ior_open:Nn` *and* `\__ior_open_stream:Nn`.)

`\ior_shell_open:Nn` `\__ior_shell_open:nN` `\__ior_shell_open:oN`

Actually much easier than either the standard open or input versions! When calling `\__kernel_ior_open:Nn` the file the pipe is added to signal a shell command, but the quotes are not added yet—they are added later by `\__kernel_file_name_quote:n`.

```
\cs_new_protected:Npn \ior_shell_open:Nn #1#2
  {
    \sys_if_shell:TF
      { \__ior_shell_open:oN { \tl_to_str:n {#2} } #1 }
      { \msg_error:nn { kernel } { pipe-failed } }
  }
\cs_new_protected:Npn \__ior_shell_open:nN #1#2
  {
    \tl_if_in:nnTF {#1} { " }
      {
        \msg_error:nne
          { kernel } { quote-in-shell } {#1}
      }
      { \__kernel_ior_open:Nn #2 { |#1 } }
  }
\cs_generate_variant:Nn \__ior_shell_open:nN { o }
\msg_new:nnnn { kernel } { pipe-failed }
  { Cannot~run~piped~system~commands. }
  {
    LaTeX~tried~to~call~a~system~process~but~this~was~not~possible.\\
    Try~the~"--shell-escape"~(or~"--enable-pipes")~option.
  }
```

(*End of definition for* `\ior_shell_open:Nn` *and* `\__ior_shell_open:nN`*. This function is documented on page* *92*.)

`\ior_close:N` `\ior_close:c`

Closing a stream means getting rid of it at the TeX level and removing from the various data structures. Unless the name passed is an invalid stream number (outside the range $[0, 15]$), it can be closed. On the other hand, it only gets added to the stack if it was not already there, to avoid duplicates building up.

```
\cs_new_protected:Npn \ior_close:N #1
  {
    \int_compare:nT { -1 < #1 < \c__ior_term_ior }
      {
        \tex_closein:D #1
        \prop_gremove:NV \g__ior_streams_prop #1
        \seq_if_in:NVF \g__ior_streams_seq #1
          { \seq_gpush:NV \g__ior_streams_seq #1 }
        \cs_gset_eq:NN #1 \c__ior_term_ior
      }
  }
\cs_generate_variant:Nn \ior_close:N { c }
```

(*End of definition for* `\ior_close:N`*. This function is documented on page* *93*.)

`\ior_show:N` `\ior_log:N` `\__ior_show:NN`

Seek the stream in the `\g__ior_streams_prop` list, then show the stream as open or closed accordingly.

```
\cs_new_protected:Npn \ior_show:N { \__ior_show:NN \tl_show:n }
\cs_generate_variant:Nn \ior_show:N { c }
\cs_new_protected:Npn \ior_log:N { \__ior_show:NN \tl_log:n }
\cs_generate_variant:Nn \ior_log:N { c }
```

```
\cs_new_protected:Npn \__ior_show:NN #1#2
  {
    \__kernel_chk_defined:NT #2
      {
        \prop_get:NVNTF \g__ior_streams_prop #2 \l__ior_internal_tl
          {
            \exp_args:Ne #1
              { \token_to_str:N #2 ~ open: ~ \l__ior_internal_tl }
          }
          { \exp_args:Ne #1 { \token_to_str:N #2 ~ closed } }
      }
  }
```

(*End of definition for* `\ior_show:N`*,* `\ior_log:N`*, and* `\__ior_show:NN`*. These functions are documented on page 93.*)

`\ior_show_list:`
`\ior_log_list:`
`\__ior_list:N`

Show the property lists, but with some "pretty printing". See the l3msg module. The first argument of the message is `ior` (as opposed to `iow`) and the second is empty if no read stream is open and non-empty (the list of streams formatted using `\msg_show_item_unbraced:nn`) otherwise. The code of the message `show-streams` takes care of translating `ior`/`iow` to English.

```
\cs_new_protected:Npn \ior_show_list: { \__ior_list:N \msg_show:nneeee }
\cs_new_protected:Npn \ior_log_list: { \__ior_list:N \msg_log:nneeee }
\cs_new_protected:Npn \__ior_list:N #1
  {
    #1 { kernel } { show-streams }
      { ior }
      {
        \prop_map_function:NN \g__ior_streams_prop
          \msg_show_item_unbraced:nn
      }
      { } { }
  }
```

(*End of definition for* `\ior_show_list:`*,* `\ior_log_list:`*, and* `\__ior_list:N`*. These functions are documented on page 93.*)

50.1.3 Reading input

`\if_eof:w`

The primitive conditional

```
\cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

(*End of definition for* `\if_eof:w`*. This function is documented on page 100.*)

`\ior_if_eof_p:N`
`\ior_if_eof:NTF`

To test if some particular input stream is exhausted the following conditional is provided. The primitive test can only deal with numbers in the range $[0, 15]$ so we catch outliers (they are exhausted).

```
\prg_new_conditional:Npnn \ior_if_eof:N #1 { p , T , F , TF }
  {
    \if_int_compare:w -1 < #1
      \if_int_compare:w #1 < \c__ior_term_ior
        \if_eof:w #1
          \prg_return_true:
        \else:
```

```
          \prg_return_false:
        \fi:
      \else:
        \prg_return_true:
      \fi:
    \else:
      \prg_return_true:
    \fi:
  }
```

(*End of definition for* `\ior_if_eof:NTF`. *This function is documented on page 96.*)

`\ior_get:NN` `\__ior_get:NN` **`\ior_get:NNTF`**

And here we read from files.

```
\cs_new_protected:Npn \ior_get:NN #1#2
  { \ior_get:NNF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_new_protected:Npn \__ior_get:NN #1#2
  { \tex_read:D #1 to #2 }
\prg_new_protected_conditional:Npnn \ior_get:NN #1#2 { T , F , TF }
  {
    \ior_if_eof:NTF #1
      { \prg_return_false: }
      {
        \__ior_get:NN #1 #2
        \prg_return_true:
      }
  }
```

(*End of definition for* `\ior_get:NN`, `\__ior_get:NN`, *and* `\ior_get:NNTF`. *These functions are documented on page 94.*)

`\ior_str_get:NN` `\__ior_str_get:NN` **`\ior_str_get:NNTF`**

Reading as strings is a more complicated wrapper, as we wish to remove the endline character and restore it afterwards.

```
\cs_new_protected:Npn \ior_str_get:NN #1#2
  { \ior_str_get:NNF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_new_protected:Npn \__ior_str_get:NN #1#2
  {
    \exp_args:Nno \use:n
      {
        \int_set:Nn \tex_endlinechar:D { -1 }
        \tex_readline:D #1 to #2
        \int_set:Nn \tex_endlinechar:D
      } { \int_use:N \tex_endlinechar:D }
  }
\prg_new_protected_conditional:Npnn \ior_str_get:NN #1#2 { T , F , TF }
  {
    \ior_if_eof:NTF #1
      { \prg_return_false: }
      {
        \__ior_str_get:NN #1 #2
        \prg_return_true:
      }
  }
```

(*End of definition for* `\ior_str_get:NN`, `\__ior_str_get:NN`, *and* `\ior_str_get:NNTF`. *These functions are documented on page 94.*)

\c__ior_term_noprompt_ior

For reading without a prompt.

```
\int_const:Nn \c__ior_term_noprompt_ior { -1 }
```

(*End of definition for* `\c__ior_term_noprompt_ior`*.*)

\ior_get_term:nN
\ior_str_get_term:nN
__ior_get_term:NnN

Getting from the terminal is better with pretty-printing.

```
\cs_new_protected:Npn \ior_get_term:nN #1#2
  { \__ior_get_term:NnN \__ior_get:NN {#1} #2 }
\cs_new_protected:Npn \ior_str_get_term:nN #1#2
  { \__ior_get_term:NnN \__ior_str_get:NN {#1} #2 }
\cs_new_protected:Npn \__ior_get_term:NnN #1#2#3
  {
    \group_begin:
      \tex_escapechar:D = -1 \scan_stop:
      \tl_if_blank:nTF {#2}
        { \exp_args:NNc #1 \c__ior_term_noprompt_ior }
        { \exp_args:NNc #1 \c__ior_term_ior }
          {#2}
    \exp_args:NNNv \group_end:
    \tl_set:Nn #3 {#2}
  }
```

(*End of definition for* `\ior_get_term:nN`*,* `\ior_str_get_term:nN`*, and* `\__ior_get_term:NnN`*. These functions are documented on page 97.*)

\ior_map_break:
\ior_map_break:n

Usual map breaking functions.

```
\cs_new:Npn \ior_map_break:
  { \prg_map_break:Nn \ior_map_break: { } }
\cs_new:Npn \ior_map_break:n
  { \prg_map_break:Nn \ior_map_break: }
```

(*End of definition for* `\ior_map_break:` *and* `\ior_map_break:n`*. These functions are documented on page 96.*)

\ior_map_inline:Nn
\ior_str_map_inline:Nn
__ior_map_inline:NNn
__ior_map_inline:NNNn
__ior_map_inline_loop:NNN

Mapping over an input stream can be done on either a token or a string basis, hence the set up. Within that, there is a check to avoid reading past the end of a file, hence the two applications of `\ior_if_eof:N` and its lower-level analogue `\if_eof:w`. This mapping cannot be nested with twice the same stream, as the stream has only one "current line".

```
\cs_new_protected:Npn \ior_map_inline:Nn
  { \__ior_map_inline:NNn \__ior_get:NN }
\cs_new_protected:Npn \ior_str_map_inline:Nn
  { \__ior_map_inline:NNn \__ior_str_get:NN }
\cs_new_protected:Npn \__ior_map_inline:NNn
  {
    \int_gincr:N \g__kernel_prg_map_int
    \exp_args:Nc \__ior_map_inline:NNNn
      { __ior_map_ \int_use:N \g__kernel_prg_map_int :n }
  }
\cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4
  {
    \cs_gset_protected:Npn #1 ##1 {#4}
    \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
    \prg_break_point:Nn \ior_map_break:
      { \int_gdecr:N \g__kernel_prg_map_int }
```

```
  }
\cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
  {
    #2 #3 \l__ior_internal_tl
    \if_eof:w #3
      \exp_after:wN \ior_map_break:
    \fi:
    \exp_args:No #1 \l__ior_internal_tl
    \__ior_map_inline_loop:NNN #1#2#3
  }
```

(*End of definition for* `\ior_map_inline:Nn` *and others. These functions are documented on page 95.*)

`\ior_map_variable:NNn`
`\ior_str_map_variable:NNn`
`\__ior_map_variable:NNNn`
`\__ior_map_variable_loop:NNNn`

Since the TeX primitive (`\read` or `\readline`) assigns the tokens read in the same way as a token list assignment, we simply call the appropriate primitive. The end-of-loop is checked using the primitive conditional for speed.

```
\cs_new_protected:Npn \ior_map_variable:NNn
  { \__ior_map_variable:NNNn \ior_get:NN }
\cs_new_protected:Npn \ior_str_map_variable:NNn
  { \__ior_map_variable:NNNn \ior_str_get:NN }
\cs_new_protected:Npn \__ior_map_variable:NNNn #1#2#3#4
  {
    \ior_if_eof:NF #2 { \__ior_map_variable_loop:NNNn #1#2#3 {#4} }
    \prg_break_point:Nn \ior_map_break: { }
  }
\cs_new_protected:Npn \__ior_map_variable_loop:NNNn #1#2#3#4
  {
    #1 #2 #3
    \if_eof:w #2
      \exp_after:wN \ior_map_break:
    \fi:
    #4
    \__ior_map_variable_loop:NNNn #1#2#3 {#4}
  }
```

(*End of definition for* `\ior_map_variable:NNn` *and others. These functions are documented on page 95.*)

50.2 Output operations

```
⟨@@=iow⟩
```

There is a lot of similarity here to the input operations, at least for many of the basics. Thus quite a bit is copied from the earlier material with minor alterations.

50.2.1 Variables and constants

`\l__iow_internal_tl`

Used as a short-term scratch variable.

```
\tl_new:N  \l__iow_internal_tl
```

(*End of definition for* `\l__iow_internal_tl`.)

`\c_log_iow`
`\c_term_iow`

Here we allocate two output streams for writing to the transcript file only (`\c_log_iow`) and to both the terminal and transcript file (`\c_term_iow`). Recent LuaTeX provide 128 write streams; we also use `\c_term_iow` as the first non-allowed write stream so its value depends on the engine.

```
\int_const:Nn \c_log_iow  { -1 }
\int_const:Nn \c_term_iow
  {
    \bool_lazy_and:nnTF
      { \sys_if_engine_luatex_p: }
      { \int_compare_p:nNn \tex_luatexversion:D > { 80 } }
      { 128 }
      { 16 }
  }
```

(*End of definition for* `\c_log_iow` *and* `\c_term_iow`*. These variables are documented on page 100.*)

`\g__iow_streams_seq` A list of the currently-available output streams to be used as a stack.

```
\seq_new:N \g__iow_streams_seq
```

(*End of definition for* `\g__iow_streams_seq`*.*)

`\l__iow_stream_tl` Used to recover the raw stream number from the stack.

```
\tl_new:N \l__iow_stream_tl
```

(*End of definition for* `\l__iow_stream_tl`*.*)

`\g__iow_streams_prop` As for reads with the appropriate adjustment of the register numbers to check on.

```
\prop_new:N \g__iow_streams_prop
\int_step_inline:nnn
  { 0 }
  {
    \cs_if_exist:NTF \contextversion
      { \tex_count:D 39 ~ }
      {
        \tex_count:D 17 ~
        \cs_if_exist:NT \loccount { - 1 }
      }
  }
  {
    \prop_gput:Nnn \g__iow_streams_prop {#1} { Reserved~by~format }
  }
```

(*End of definition for* `\g__iow_streams_prop`*.*)

50.2.2 Internal auxiliaries

`\s__iow_mark`
`\s__iow_stop` Internal scan marks.

```
\scan_new:N \s__iow_mark
\scan_new:N \s__iow_stop
```

(*End of definition for* `\s__iow_mark` *and* `\s__iow_stop`*.*)

`\__iow_use_i_delimit_by_s_stop:nw` Functions to gobble up to a scan mark.

```
\cs_new:Npn \__iow_use_i_delimit_by_s_stop:nw #1 #2 \s__iow_stop {#1}
```

(*End of definition for* `\__iow_use_i_delimit_by_s_stop:nw`*.*)

\q__iow_nil Internal quarks.

```
\quark_new:N \q__iow_nil
```

(*End of definition for* \q__iow_nil*.*)

50.3 Stream management

\iow_new:N
\iow_new:c

Reserving a new stream is done by defining the name as equal to writing to the terminal: odd but at least consistent.

```
\cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
\cs_generate_variant:Nn \iow_new:N { c }
```

(*End of definition for* \iow_new:N*. This function is documented on page* 92*.*)

\g_tmpa_iow
\g_tmpb_iow

The usual scratch space.

```
\iow_new:N \g_tmpa_iow
\iow_new:N \g_tmpb_iow
```

(*End of definition for* \g_tmpa_iow *and* \g_tmpb_iow*. These variables are documented on page* 100*.*)

__iow_new:N As for read streams, copy \newwrite, making sure that it is not \outer. For ConTEXt, we have to deal with the fact that \newwrite works like our own: it actually checks before altering definition.

```
\exp_args:NNf \cs_new_protected:Npn \__iow_new:N
  { \exp_args:NNc \exp_after:wN \exp_stop_f: { newwrite } }
\cs_if_exist:NT \contextversion
  {
    \cs_new_eq:NN \__iow_new_aux:N \__iow_new:N
    \cs_gset_protected:Npn \__iow_new:N #1
      {
        \cs_undefine:N #1
        \__iow_new_aux:N #1
      }
  }
```

(*End of definition for* __iow_new:N*.*)

\l__iow_file_name_tl Data storage.

```
\tl_new:N \l__iow_file_name_tl
```

(*End of definition for* \l__iow_file_name_tl*.*)

\iow_open:Nn
\iow_open:NV
\iow_open:cn
\iow_open:cV
__kernel_iow_open:Nn
__kernel_iow_open:No
__iow_open_stream:Nn
__iow_open_stream:NV

The same idea as for reading, but without the path and without the need to allow for a conditional version.

```
\cs_new_protected:Npn \iow_open:Nn #1#2
  {
    \__kernel_tl_set:Nx \l__iow_file_name_tl
      { \__kernel_file_name_sanitize:n {#2} }
    \__kernel_iow_open:No #1 \l__iow_file_name_tl
  }
\cs_generate_variant:Nn \iow_open:Nn { NV , c , cV }
\cs_new_protected:Npn \__kernel_iow_open:Nn #1#2
  {
```

```
    \iow_close:N #1
    \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
      { \__iow_open_stream:Nn #1 {#2} }
      {
        \__iow_new:N #1
        \__kernel_tl_set:Nx \l__iow_stream_tl { \int_eval:n {#1} }
        \__iow_open_stream:Nn #1 {#2}
      }
  }
\cs_generate_variant:Nn \__kernel_iow_open:Nn { No }
\cs_new_protected:Npn \__iow_open_stream:Nn #1#2
  {
    \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:
    \prop_gput:NVn \g__iow_streams_prop #1 {#2}
    \tex_immediate:D \tex_openout:D
        #1 \__kernel_file_name_quote:n {#2} \scan_stop:
  }
\cs_generate_variant:Nn \__iow_open_stream:Nn { NV }
```

(*End of definition for* `\iow_open:Nn`, `\__kernel_iow_open:Nn`, *and* `\__iow_open_stream:Nn`. *This function is documented on page 92.*)

`\iow_shell_open:Nn`
`\__iow_shell_open:nN`
`\__iow_shell_open:oN`

Very similar to the `ior` version

```
\cs_new_protected:Npn \iow_shell_open:Nn #1#2
  {
    \sys_if_shell:TF
      { \__iow_shell_open:oN { \tl_to_str:n {#2} } #1 }
      { \msg_error:nn { kernel } { pipe-failed } }
  }
\cs_new_protected:Npn \__iow_shell_open:nN #1#2
  {
    \tl_if_in:nnTF {#1} { " }
      {
        \msg_error:nne
          { kernel } { quote-in-shell } {#1}
      }
      { \__kernel_iow_open:Nn #2 { |#1 } }
  }
\cs_generate_variant:Nn \__iow_shell_open:nN { o }
```

(*End of definition for* `\iow_shell_open:Nn` *and* `\__iow_shell_open:nN`. *This function is documented on page 92.*)

`\iow_close:N`
`\iow_close:c`

Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.

```
\cs_new_protected:Npn \iow_close:N #1
  {
    \int_compare:nT { \c_log_iow < #1 < \c_term_iow }
      {
        \tex_immediate:D \tex_closeout:D #1
        \prop_gremove:NV \g__iow_streams_prop #1
        \seq_if_in:NVF \g__iow_streams_seq #1
          { \seq_gpush:NV \g__iow_streams_seq #1 }
        \cs_gset_eq:NN #1 \c_term_iow
```

```
      }
  }
\cs_generate_variant:Nn \iow_close:N { c }
```

(*End of definition for* `\iow_close:N`. *This function is documented on page 93.*)

`\iow_show:N` `\iow_log:N` `\__iow_show:NN`

Seek the stream in the `\g__iow_streams_prop` list, then show the stream as open or closed accordingly.

```
\cs_new_protected:Npn \iow_show:N { \__iow_show:NN \tl_show:n }
\cs_generate_variant:Nn \iow_show:N { c }
\cs_new_protected:Npn \iow_log:N { \__iow_show:NN \tl_log:n }
\cs_generate_variant:Nn \iow_log:N { c }
\cs_new_protected:Npn \__iow_show:NN #1#2
  {
    \__kernel_chk_defined:NT #2
      {
        \prop_get:NVNTF \g__iow_streams_prop #2 \l__iow_internal_tl
          {
            \exp_args:Ne #1
              { \token_to_str:N #2 ~ open: ~ \l__iow_internal_tl }
          }
          { \exp_args:Ne #1 { \token_to_str:N #2 ~ closed } }
      }
  }
```

(*End of definition for* `\iow_show:N`, `\iow_log:N`, *and* `\__iow_show:NN`. *These functions are documented on page 93.*)

`\iow_show_list:` `\iow_log_list:` `\__iow_list:N`

Done as for input, but with a copy of the auxiliary so the name is correct.

```
\cs_new_protected:Npn \iow_show_list: { \__iow_list:N \msg_show:nneeee }
\cs_new_protected:Npn \iow_log_list: { \__iow_list:N \msg_log:nneeee }
\cs_new_protected:Npn \__iow_list:N #1
  {
    #1 { kernel } { show-streams }
      { iow }
      {
        \prop_map_function:NN \g__iow_streams_prop
          \msg_show_item_unbraced:nn
      }
      { } { }
  }
```

(*End of definition for* `\iow_show_list:`, `\iow_log_list:`, *and* `\__iow_list:N`. *These functions are documented on page 93.*)

50.3.1 Deferred writing

`\iow_shipout_e:Nn` `\iow_shipout_e:Ne` `\iow_shipout_e:cn` `\iow_shipout_e:ce`

First the easy part, this is the primitive, which expects its argument to be braced.

```
\cs_new_protected:Npn \iow_shipout_e:Nn #1#2
  { \tex_write:D #1 {#2} }
\cs_generate_variant:Nn \iow_shipout_e:Nn { Ne , c, ce }
```

(*End of definition for* `\iow_shipout_e:Nn`. *This function is documented on page 98.*)

\iow_shipout:Nn
\iow_shipout:Ne
\iow_shipout:Nx
\iow_shipout:cn
\iow_shipout:ce
\iow_shipout:cx

With ε-TeX available deferred writing without expansion is easy.

```
\cs_new_protected:Npn \iow_shipout:Nn #1#2
  { \tex_write:D #1 { \exp_not:n {#2} } }
\cs_generate_variant:Nn \iow_shipout:Nn { Ne , c, ce }
\cs_generate_variant:Nn \iow_shipout:Nn { Nx , cx }
```

(*End of definition for* `\iow_shipout:Nn`. *This function is documented on page 97.*)

50.3.2 Immediate writing

__kernel_iow_with:Nnn
__iow_with:nNnn
__iow_with:oNnn

If the integer `#1` is equal to `#2`, just leave `#3` in the input stream. Otherwise, pass the old value to an auxiliary, which sets the integer to the new value, runs the code, and restores the integer.

```
\cs_new_protected:Npn \__kernel_iow_with:Nnn #1#2
  {
    \int_compare:nNnTF {#1} = {#2}
      { \use:n }
      { \__iow_with:oNnn { \int_use:N #1 } #1 {#2} }
  }
\cs_new_protected:Npn \__iow_with:nNnn #1#2#3#4
  {
    \int_set:Nn #2 {#3}
    #4
    \int_set:Nn #2 {#1}
  }
\cs_generate_variant:Nn \__iow_with:nNnn { o }
```

(*End of definition for* `\__kernel_iow_with:Nnn` *and* `\__iow_with:nNnn`.)

\iow_now:Nn
\iow_now:NV
\iow_now:Ne
\iow_now:Nx
\iow_now:cn
\iow_now:cV
\iow_now:ce
\iow_now:cx

This routine writes the second argument onto the output stream without expansion. If this stream isn't open, the output goes to the terminal instead. If the first argument is no output stream at all, we get an internal error. We don't use the expansion done by `\write` to get the `Nx` variant, because it differs in subtle ways from x-expansion, namely, macro parameter characters would not need to be doubled. We set the `\newlinechar` to 10 using `\__kernel_iow_with:Nnn` to support formats such as plain TeX: otherwise, `\iow_newline:` would not work. We do not do this for `\iow_shipout:Nn` or `\iow_shipout_x:Nn`, as TeX looks at the value of the `\newlinechar` at shipout time in those cases.

```
\cs_new_protected:Npn \iow_now:Nn #1#2
  {
    \__kernel_iow_with:Nnn \tex_newlinechar:D { `\^^J }
      { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
  }
\cs_generate_variant:Nn \iow_now:Nn { NV , Ne , c , cV , ce }
\cs_generate_variant:Nn \iow_now:Nn { Nx , cx }
```

(*End of definition for* `\iow_now:Nn`. *This function is documented on page 97.*)

\iow_log:n
\iow_log:e
\iow_log:x
\iow_term:n
\iow_term:e
\iow_term:x

Writing to the log and the terminal directly are relatively easy; as we need the two e-type variants for bootstrapping, they are redefinitions here.

```
\cs_new_protected:Npn \iow_log:n  { \iow_now:Nn \c_log_iow  }
\cs_set_protected:Npn \iow_log:e  { \iow_now:Ne \c_log_iow  }
\cs_generate_variant:Nn \iow_log:n { x }
```

```
\cs_new_protected:Npn \iow_term:n { \iow_now:Nn \c_term_iow }
\cs_set_protected:Npn \iow_term:e { \iow_now:Ne \c_term_iow }
\cs_generate_variant:Nn \iow_term:n { x }
```

(*End of definition for* `\iow_log:n` *and* `\iow_term:n`*. These functions are documented on page 97.*)

50.3.3 Special characters for writing

`\iow_newline:` Global variable holding the character that forces a new line when something is written to an output stream.

```
\cs_new:Npn \iow_newline: { ^^J }
```

(*End of definition for* `\iow_newline:`*. This function is documented on page 98.*)

`\iow_char:N` Function to write any escaped char to an output stream.

```
\cs_new_eq:NN \iow_char:N \cs_to_str:N
```

(*End of definition for* `\iow_char:N`*. This function is documented on page 98.*)

50.3.4 Hard-wrapping lines to a character count

The code here implements a generic hard-wrapping function. This is used by the messaging system, but is designed such that it is available for other uses.

`\l_iow_line_count_int` This is the “raw” number of characters in a line which can be written to the terminal. The standard value is the line length typically used by TEX Live and MiKTEX.

```
\int_new:N  \l_iow_line_count_int
\int_set:Nn \l_iow_line_count_int { 78 }
```

(*End of definition for* `\l_iow_line_count_int`*. This variable is documented on page 100.*)

`\l__iow_newline_tl` The token list inserted to produce a new line, with the ⟨*run-on text*⟩.

```
\tl_new:N \l__iow_newline_tl
```

(*End of definition for* `\l__iow_newline_tl`*.*)

`\l__iow_line_target_int` This stores the target line count: the full number of characters in a line, minus any part for a leader at the start of each line.

```
\int_new:N \l__iow_line_target_int
```

(*End of definition for* `\l__iow_line_target_int`*.*)

`\__iow_set_indent:n`
`\__iow_unindent:w`
`\l__iow_one_indent_tl`
`\l__iow_one_indent_int`

The `one_indent` variables hold one indentation marker and its length. The `\__iow_unindent:w` auxiliary removes one indentation. The function `\__iow_set_indent:n` (that could possibly be public) sets the indentation in a consistent way. We set it to four spaces by default.

```
\tl_new:N \l__iow_one_indent_tl
\int_new:N \l__iow_one_indent_int
\cs_new:Npn \__iow_unindent:w { }
\cs_new_protected:Npn \__iow_set_indent:n #1
  {
    \__kernel_tl_set:Nx \l__iow_one_indent_tl
      { \exp_args:No \__kernel_str_to_other_fast:n { \tl_to_str:n {#1} } }
    \int_set:Nn \l__iow_one_indent_int
```

```
      { \str_count:N \l__iow_one_indent_tl }
    \exp_last_unbraced:NNo
      \cs_set:Npn \__iow_unindent:w \l__iow_one_indent_tl { }
  }
\exp_args:Ne \__iow_set_indent:n { \prg_replicate:nn { 4 } { ~ } }
```

(*End of definition for* `\__iow_set_indent:n` *and others.*)

`\l__iow_indent_tl`
`\l__iow_indent_int`

The current indentation (some copies of `\l__iow_one_indent_tl`) and its number of characters.

```
\tl_new:N \l__iow_indent_tl
\int_new:N \l__iow_indent_int
```

(*End of definition for* `\l__iow_indent_tl` *and* `\l__iow_indent_int`*.*)

`\l__iow_line_tl`
`\l__iow_line_part_tl`

These hold the current line of text and a partial line to be added to it, respectively.

```
\tl_new:N \l__iow_line_tl
\tl_new:N \l__iow_line_part_tl
```

(*End of definition for* `\l__iow_line_tl` *and* `\l__iow_line_part_tl`*.*)

`\l__iow_line_break_bool`

Indicates whether the line was broken precisely at a chunk boundary.

```
\bool_new:N \l__iow_line_break_bool
```

(*End of definition for* `\l__iow_line_break_bool`*.*)

`\l__iow_wrap_tl`

Used for the expansion step before detokenizing, and for the output from wrapping text: fully expanded and with lines which are not overly long.

```
\tl_new:N \l__iow_wrap_tl
```

(*End of definition for* `\l__iow_wrap_tl`*.*)

`\c__iow_wrap_marker_tl`
`\c__iow_wrap_end_marker_tl`
`\c__iow_wrap_newline_marker_tl`
`\c__iow_wrap_allow_break_marker_tl`
`\c__iow_wrap_indent_marker_tl`
`\c__iow_wrap_unindent_marker_tl`

Every special action of the wrapping code is starts with the same recognizable string, `\c__iow_wrap_marker_tl`. Upon seeing that "word", the wrapping code reads one space-delimited argument to know what operation to perform. The setting of `\escapechar` here is not very important, but makes `\c__iow_wrap_marker_tl` look marginally nicer.

```
\group_begin:
  \int_set:Nn \tex_escapechar:D { -1 }
  \tl_const:Ne \c__iow_wrap_marker_tl
    { \tl_to_str:n { \^^I \^^O \^^W \^^_ \^^W \^^R \^^A \^^P } }
\group_end:
\tl_map_inline:nn
  { { end } { newline } { allow_break } { indent } { unindent } }
  {
    \tl_const:ce { c__iow_wrap_ #1 _marker_tl }
      {
        \c__iow_wrap_marker_tl
        #1
        \c_catcode_other_space_tl
      }
  }
```

(*End of definition for* `\c__iow_wrap_marker_tl` *and others.*)

\iow_wrap_allow_break:
__iow_wrap_allow_break:
__iow_wrap_allow_break_error:

We set \iow_wrap_allow_break:n to produce an error when outside messages. Within wrapped message, it is set to __iow_wrap_allow_break: when valid and otherwise to __iow_wrap_allow_break_error:. The second produces an error expandably.

```
\cs_new_protected:Npn \iow_wrap_allow_break:
  {
    \msg_error:nnnn { kernel } { iow-indent }
      { \iow_wrap:nnnN } { \iow_wrap_allow_break: }
  }
\cs_new:Npe \__iow_wrap_allow_break: { \c__iow_wrap_allow_break_marker_tl }
\cs_new:Npn \__iow_wrap_allow_break_error:
  {
    \msg_expandable_error:nnnn { kernel } { iow-indent }
      { \iow_wrap:nnnN } { \iow_wrap_allow_break: }
  }
```

(*End of definition for* \iow_wrap_allow_break: , __iow_wrap_allow_break: , *and* __iow_wrap_allow_break_error:. *This function is documented on page 99.*)

\iow_indent:n
__iow_indent:n
__iow_indent_error:n

We set \iow_indent:n to produce an error when outside messages. Within wrapped message, it is set to __iow_indent:n when valid and otherwise to __iow_indent_error:n. The first places the instruction for increasing the indentation before its argument, and the instruction for unindenting afterwards. The second produces an error expandably. Note that there are no forced line-break, so the indentation only changes when the next line is started.

```
\cs_new_protected:Npn \iow_indent:n #1
  {
    \msg_error:nnnnn { kernel } { iow-indent }
      { \iow_wrap:nnnN } { \iow_indent:n } {#1}
    #1
  }
\cs_new:Npe \__iow_indent:n #1
  {
    \c__iow_wrap_indent_marker_tl
    #1
    \c__iow_wrap_unindent_marker_tl
  }
\cs_new:Npn \__iow_indent_error:n #1
  {
    \msg_expandable_error:nnnnn { kernel } { iow-indent }
      { \iow_wrap:nnnN } { \iow_indent:n } {#1}
    #1
  }
```

(*End of definition for* \iow_indent:n , __iow_indent:n , *and* __iow_indent_error:n. *This function is documented on page 99.*)

\iow_wrap:nnnN
\iow_wrap:nenN

The main wrapping function works as follows. First give \\, \␣ and other formatting commands the correct definition for messages and perform the given setup #3. The definition of \␣ uses an "other" space rather than a normal space, because the latter might be absorbed by TeX to end a number or other f-type expansions. Use \conditionally@traceoff if defined; it is introduced by the trace package and suppresses uninteresting tracing of the wrapping code.

```
\cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
```

```
  {
    \group_begin:
      \cs_if_exist_use:N \conditionally@traceoff
      \int_set:Nn \tex_escapechar:D { -1 }
      \cs_set:Npe \{ { \token_to_str:N \{ }
      \cs_set:Npe \# { \token_to_str:N \# }
      \cs_set:Npe \} { \token_to_str:N \} }
      \cs_set:Npe \% { \token_to_str:N \% }
      \cs_set:Npe \~ { \token_to_str:N \~ }
      \int_set:Nn \tex_escapechar:D { 92 }
      \cs_set_eq:NN \\ \iow_newline:
      \cs_set_eq:NN \  \c_catcode_other_space_tl
      \cs_set_eq:NN \iow_wrap_allow_break: \__iow_wrap_allow_break:
      \cs_set_eq:NN \iow_indent:n \__iow_indent:n
      #3
```

Then fully-expand the input: in package mode, the expansion uses LaTeX 2ε's `\protect` mechanism in the same way as `\typeout`. In generic mode this setting is useless but harmless. As soon as the expansion is done, reset `\iow_indent:n` to its error definition: it only works in the first argument of `\iow_wrap:nnnN`.

```
      \cs_set_eq:NN \protect \token_to_str:N
      \__kernel_tl_set:Nx \l__iow_wrap_tl {#1}
      \cs_set_eq:NN \iow_wrap_allow_break: \__iow_wrap_allow_break_error:
      \cs_set_eq:NN \iow_indent:n \__iow_indent_error:n
```

Afterwards, set the newline marker (two assignments to fully expand, then convert to a string) and initialize the target count for lines (the first line has target count `\l_iow_line_count_int` instead).

```
      \__kernel_tl_set:Nx \l__iow_newline_tl { \iow_newline: #2 }
      \__kernel_tl_set:Nx \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
      \int_set:Nn \l__iow_line_target_int
        { \l_iow_line_count_int - \str_count:N \l__iow_newline_tl + 1 }
```

Sanity check.

```
      \int_compare:nNnT { \l__iow_line_target_int } < 0
        {
          \tl_set:Nn \l__iow_newline_tl { \iow_newline: }
          \int_set:Nn \l__iow_line_target_int
            { \l_iow_line_count_int + 1 }
        }
```

There is then a loop over the input, which stores the wrapped result in `\l__iow_wrap_tl`. After the loop, the resulting text is passed on to the function which has been given as a post-processor. The `\tl_to_str:N` step converts the "other" spaces back to normal spaces. The `f`-expansion removes a leading space from `\l__iow_wrap_tl`.

```
      \__iow_wrap_do:
    \exp_args:NNf \group_end:
    #4 { \tl_to_str:N \l__iow_wrap_tl }
  }
\cs_generate_variant:Nn \iow_wrap:nnnN { ne }
```

(*End of definition for* `\iow_wrap:nnnN`. *This function is documented on page 99.*)

`\__iow_wrap_do:`
`\__iow_wrap_fix_newline:w`
`\__iow_wrap_start:w`

Escape spaces and change newlines to `\c__iow_wrap_newline_marker_tl`. Set up a few variables, in particular the initial value of `\l__iow_wrap_tl`: the space stops the f-expansion of the main wrapping function and \use_none:n removes a newline marker inserted by later code. The main loop consists of repeatedly calling the chunk auxiliary to wrap chunks delimited by (newline or indentation) markers.

```
\cs_new_protected:Npn \__iow_wrap_do:
  {
    \__kernel_tl_set:Nx \l__iow_wrap_tl
      {
        \exp_args:No \__kernel_str_to_other_fast:n \l__iow_wrap_tl
        \c__iow_wrap_end_marker_tl
      }
    \__kernel_tl_set:Nx \l__iow_wrap_tl
      {
        \exp_after:wN \__iow_wrap_fix_newline:w \l__iow_wrap_tl
          ^^J \q__iow_nil ^^J \s__iow_stop
      }
    \exp_after:wN \__iow_wrap_start:w \l__iow_wrap_tl
  }
\cs_new:Npn \__iow_wrap_fix_newline:w #1 ^^J #2 ^^J
  {
    #1
    \if_meaning:w \q__iow_nil #2
      \__iow_use_i_delimit_by_s_stop:nw
    \fi:
    \c__iow_wrap_newline_marker_tl
    \__iow_wrap_fix_newline:w #2 ^^J
  }
\cs_new_protected:Npn \__iow_wrap_start:w
  {
    \bool_set_false:N \l__iow_line_break_bool
    \tl_clear:N \l__iow_line_tl
    \tl_clear:N \l__iow_line_part_tl
    \tl_set:Nn \l__iow_wrap_tl { ~ \use_none:n }
    \int_zero:N \l__iow_indent_int
    \tl_clear:N \l__iow_indent_tl
    \__iow_wrap_chunk:nw { \l_iow_line_count_int }
  }
```

(*End of definition for* __iow_wrap_do:*,* __iow_wrap_fix_newline:w*, and* __iow_wrap_start:w*.*)

__iow_wrap_chunk:nw
__iow_wrap_next:nw

The chunk and next auxiliaries are defined indirectly to obtain the expansions of \c_-catcode_other_space_tl and \c__iow_wrap_marker_tl in their definition. The next auxiliary calls a function corresponding to the type of marker (its ##2), which can be newline or indent or unindent or end. The first argument of the chunk auxiliary is a target number of characters and the second is some string to wrap. If the chunk is empty simply call next. Otherwise, set up a call to __iow_wrap_line:nw, including the indentation if the current line is empty, and including a trailing space (#1) before the __iow_wrap_end_chunk:w auxiliary.

```
\cs_set_protected:Npn \__iow_tmp:w #1#2
  {
    \cs_new_protected:Npn \__iow_wrap_chunk:nw ##1##2 #2
      {
        \tl_if_empty:nTF {##2}
          {
```

```
            \tl_clear:N \l__iow_line_part_tl
            \__iow_wrap_next:nw {##1}
          }
          {
            \tl_if_empty:NTF \l__iow_line_tl
              {
                \__iow_wrap_line:nw
                  { \l__iow_indent_tl }
                  ##1 - \l__iow_indent_int ;
              }
              { \__iow_wrap_line:nw { } ##1 ; }
            ##2 #1
            \__iow_wrap_end_chunk:w 7 6 5 4 3 2 1 0 \s__iow_stop
          }
      }
    \cs_new_protected:Npn \__iow_wrap_next:nw ##1##2 #1
      { \use:c { __iow_wrap_##2:n } {##1} }
  }
\exp_args:NVV \__iow_tmp:w \c_catcode_other_space_tl \c__iow_wrap_marker_tl
```

(*End of definition for* `\__iow_wrap_chunk:nw` *and* `\__iow_wrap_next:nw`.)

`\__iow_wrap_line:nw`
`\__iow_wrap_line_loop:w`
`\__iow_wrap_line_aux:Nw`
`\__iow_wrap_line_seven:nnnnnnn`
`\__iow_wrap_line_end:NnnnnnnnN`
`\__iow_wrap_line_end:nw`
`\__iow_wrap_end_chunk:w`

This is followed by `{`⟨*string*⟩`}` ⟨*int expr*⟩ `;`. It stores the ⟨*string*⟩ and up to ⟨*int expr*⟩ characters from the current chunk into `\l__iow_line_part_tl`. Characters are grabbed 8 at a time and left in `\l__iow_line_part_tl` by the `line_loop` auxiliary. When $k < 8$ remain to be found, the `line_aux` auxiliary calls the `line_end` auxiliary followed by (the single digit) k, then $7 - k$ empty brace groups, then the chunk's remaining characters. The `line_end` auxiliary leaves k characters from the chunk in the line part, then ends the assignment. Ignore the `\use_none:nnnnn` line for now. If the next character is a space the line can be broken there: store what we found into the result and get the next line. Otherwise some work is needed to find a break-point. So far we have ignored what happens if the chunk is shorter than the requested number of characters: this is dealt with by the `end_chunk` auxiliary, which gets treated like a character by the rest of the code. It ends up being called either as one of the arguments `#2`–`#9` of the `line_loop` auxiliary or as one of the arguments `#2`–`#8` of the `line_end` auxiliary. In both cases stop the assignment and work out how many characters are still needed. Notice that when we have exactly seven arguments to clean up, a `\exp_stop_f:` has to be inserted to stop the `\exp:w`. The weird `\use_none:nnnnn` ensures that the required data is in the right place.

```
\cs_new_protected:Npn \__iow_wrap_line:nw #1
  {
    \tex_edef:D \l__iow_line_part_tl { \if_false: } \fi:
    #1
    \exp_after:wN \__iow_wrap_line_loop:w
    \int_value:w \int_eval:w
  }
\cs_new:Npn \__iow_wrap_line_loop:w #1 ; #2#3#4#5#6#7#8#9
  {
    \if_int_compare:w #1 < 8 \exp_stop_f:
      \__iow_wrap_line_aux:Nw #1
    \fi:
    #2 #3 #4 #5 #6 #7 #8 #9
    \exp_after:wN \__iow_wrap_line_loop:w
```

```
    \int_value:w \int_eval:w #1 - 8 ;
  }
\cs_new:Npn \__iow_wrap_line_aux:Nw #1#2#3 \exp_after:wN #4 ;
  {
    #2
    \exp_after:wN \__iow_wrap_line_end:NnnnnnnnN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
    \exp_after:wN \exp_after:wN
    \if_case:w #1 \exp_stop_f:
         \prg_do_nothing:
    \or: \use_none:n
    \or: \use_none:nn
    \or: \use_none:nnn
    \or: \use_none:nnnn
    \or: \use_none:nnnnn
    \or: \use_none:nnnnnn
    \or: \__iow_wrap_line_seven:nnnnnnn
    \fi:
    { } { } { } { } { } { } { } #3
  }
\cs_new:Npn \__iow_wrap_line_seven:nnnnnnn #1#2#3#4#5#6#7 { \exp_stop_f: }
\cs_new:Npn \__iow_wrap_line_end:NnnnnnnnN #1#2#3#4#5#6#7#8#9
  {
    #2 #3 #4 #5 #6 #7 #8
    \use_none:nnnnn \int_eval:w 8 - ; #9
    \token_if_eq_charcode:NNTF \c_space_token #9
      { \__iow_wrap_line_end:nw { } }
      { \if_false: { \fi: } \__iow_wrap_break:w #9 }
  }
\cs_new:Npn \__iow_wrap_line_end:nw #1
  {
    \if_false: { \fi: }
    \__iow_wrap_store_do:n {#1}
    \__iow_wrap_next_line:w
  }
\cs_new:Npn \__iow_wrap_end_chunk:w
    #1 \int_eval:w #2 - #3 ; #4#5 \s__iow_stop
  {
    \if_false: { \fi: }
    \exp_args:Nf \__iow_wrap_next:nw { \int_eval:n { #2 - #4 } }
  }
```

(*End of definition for* `\__iow_wrap_line:nw` *and others.*)

`\__iow_wrap_break:w`
`\__iow_wrap_break_first:w`
`\__iow_wrap_break_none:w`
`\__iow_wrap_break_loop:w`
`\__iow_wrap_break_end:w`

Functions here are defined indirectly: `\__iow_tmp:w` is eventually called with an "other" space as its argument. The goal is to remove from `\l__iow_line_part_tl` the part after the last space. In most cases this is done by repeatedly calling the `break_loop` auxiliary, which leaves "words" (delimited by spaces) until it hits the trailing space: then its argument `##3` is `? \__iow_wrap_break_end:w` instead of a single token, and that `break_end` auxiliary leaves in the assignment the line until the last space, then calls `\__iow_wrap_line_end:nw` to finish up the line and move on to the next. If there is no space in `\l__iow_line_part_tl` then the `break_first` auxiliary calls the `break_none` auxiliary. In that case, if the current line is empty, the complete word (including

##4, characters beyond what we had grabbed) is added to the line, making it over-long. Otherwise, the word is used for the following line (and the last space of the line so far is removed because it was inserted due to the presence of a marker).

```
\cs_set_protected:Npn \__iow_tmp:w #1
  {
    \cs_new:Npn \__iow_wrap_break:w
      {
        \tex_edef:D \l__iow_line_part_tl
          { \if_false: } \fi:
            \exp_after:wN \__iow_wrap_break_first:w
            \l__iow_line_part_tl
            #1
            { ? \__iow_wrap_break_end:w }
            \s__iow_mark
      }
    \cs_new:Npn \__iow_wrap_break_first:w ##1 #1 ##2
      {
        \use_none:nn ##2 \__iow_wrap_break_none:w
        \__iow_wrap_break_loop:w ##1 #1 ##2
      }
    \cs_new:Npn \__iow_wrap_break_none:w ##1##2 #1 ##3 \s__iow_mark ##4 #1
      {
        \tl_if_empty:NTF \l__iow_line_tl
          { ##2 ##4 \__iow_wrap_line_end:nw { } }
          { \__iow_wrap_line_end:nw { \__iow_wrap_trim:N } ##2 ##4 #1 }
      }
    \cs_new:Npn \__iow_wrap_break_loop:w ##1 #1 ##2 #1 ##3
      {
        \use_none:n ##3
        ##1 #1
        \__iow_wrap_break_loop:w ##2 #1 ##3
      }
    \cs_new:Npn \__iow_wrap_break_end:w ##1 #1 ##2 ##3 #1 ##4 \s__iow_mark
      { ##1 \__iow_wrap_line_end:nw { } ##3 }
  }
\exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl
```

(*End of definition for* `\__iow_wrap_break:w` *and others.*)

`\__iow_wrap_next_line:w` The special case where the end of a line coincides with the end of a chunk is detected here, to avoid a spurious empty line. Otherwise, call `\__iow_wrap_line:nw` to find characters for the next line (remembering to account for the indentation).

```
\cs_new_protected:Npn \__iow_wrap_next_line:w #1#2 \s__iow_stop
  {
    \tl_clear:N \l__iow_line_tl
    \token_if_eq_meaning:NNTF #1 \__iow_wrap_end_chunk:w
      {
        \tl_clear:N \l__iow_line_part_tl
        \bool_set_true:N \l__iow_line_break_bool
        \__iow_wrap_next:nw { \l__iow_line_target_int }
      }
      {
        \__iow_wrap_line:nw
          { \l__iow_indent_tl }
```

```
          \l__iow_line_target_int - \l__iow_indent_int ;
          #1 #2 \s__iow_stop
        }
    }
```

(*End of definition for* `\__iow_wrap_next_line:w`.)

`\__iow_wrap_allow_break:n`

This is called after a chunk has been wrapped. The `\l__iow_line_part_tl` typically ends with a space (except at the beginning of a line?), which we remove since the `allow_-break` marker should not insert a space. Then move on with the next chunk, making sure to adjust the target number of characters for the line in case we did remove a space.

```
\cs_new_protected:Npn \__iow_wrap_allow_break:n #1
  {
    \__kernel_tl_set:Nx \l__iow_line_tl
      { \l__iow_line_tl \__iow_wrap_trim:N \l__iow_line_part_tl }
    \bool_set_false:N \l__iow_line_break_bool
    \tl_if_empty:NTF \l__iow_line_part_tl
      { \__iow_wrap_chunk:nw {#1} }
      { \exp_args:Nf \__iow_wrap_chunk:nw { \int_eval:n { #1 + 1 } } }
  }
```

(*End of definition for* `\__iow_wrap_allow_break:n`.)

`\__iow_wrap_indent:n`
`\__iow_wrap_unindent:n`

These functions are called after a chunk has been wrapped, when encountering `indent`/`unindent` markers. Add the line part (last line part of the previous chunk) to the line so far and reset a boolean denoting the presence of a line-break. Most importantly, add or remove one indent from the current indent (both the integer and the token list). Finally, continue wrapping.

```
\cs_new_protected:Npn \__iow_wrap_indent:n #1
  {
    \tl_put_right:Ne \l__iow_line_tl { \l__iow_line_part_tl }
    \bool_set_false:N \l__iow_line_break_bool
    \int_add:Nn \l__iow_indent_int { \l__iow_one_indent_int }
    \tl_put_right:No \l__iow_indent_tl { \l__iow_one_indent_tl }
    \__iow_wrap_chunk:nw {#1}
  }
\cs_new_protected:Npn \__iow_wrap_unindent:n #1
  {
    \tl_put_right:Ne \l__iow_line_tl { \l__iow_line_part_tl }
    \bool_set_false:N \l__iow_line_break_bool
    \int_sub:Nn \l__iow_indent_int { \l__iow_one_indent_int }
    \__kernel_tl_set:Nx \l__iow_indent_tl
      { \exp_after:wN \__iow_unindent:w \l__iow_indent_tl }
    \__iow_wrap_chunk:nw {#1}
  }
```

(*End of definition for* `\__iow_wrap_indent:n` *and* `\__iow_wrap_unindent:n`.)

`\__iow_wrap_newline:n`
`\__iow_wrap_end:n`

These functions are called after a chunk has been line-wrapped, when encountering a `newline`/`end` marker. Unless we just took a line-break, store the line part and the line so far into the whole `\l__iow_wrap_tl`, trimming a trailing space. In the `newline` case look for a new line (of length `\l__iow_line_target_int`) in a new chunk.

```
\cs_new_protected:Npn \__iow_wrap_newline:n #1
  {
```

```
    \bool_if:NF \l__iow_line_break_bool
      { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
    \bool_set_false:N \l__iow_line_break_bool
    \__iow_wrap_chunk:nw { \l__iow_line_target_int }
  }
\cs_new_protected:Npn \__iow_wrap_end:n #1
  {
    \bool_if:NF \l__iow_line_break_bool
      { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
    \bool_set_false:N \l__iow_line_break_bool
  }
```

(*End of definition for* `\__iow_wrap_newline:n` *and* `\__iow_wrap_end:n`*.*)

`\__iow_wrap_store_do:n`

First add the last line part to the line, then append it to `\l__iow_wrap_tl` with the appropriate new line (with "run-on" text), possibly with its last space removed (`#1` is empty or `\__iow_wrap_trim:N`).

```
\cs_new_protected:Npn \__iow_wrap_store_do:n #1
  {
    \__kernel_tl_set:Nx \l__iow_line_tl
      { \l__iow_line_tl \l__iow_line_part_tl }
    \__kernel_tl_set:Nx \l__iow_wrap_tl
      {
        \l__iow_wrap_tl
        \l__iow_newline_tl
        #1 \l__iow_line_tl
      }
    \tl_clear:N \l__iow_line_tl
  }
```

(*End of definition for* `\__iow_wrap_store_do:n`*.*)

`\__iow_wrap_trim:N`
`\__iow_wrap_trim:w`
`\__iow_wrap_trim_aux:w`

Remove one trailing "other" space from the argument if present.

```
\cs_set_protected:Npn \__iow_tmp:w #1
  {
    \cs_new:Npn \__iow_wrap_trim:N ##1
      { \exp_after:wN \__iow_wrap_trim:w ##1 \s__iow_mark #1 \s__iow_mark \s__iow_stop }
    \cs_new:Npn \__iow_wrap_trim:w ##1 #1 \s__iow_mark
      { \__iow_wrap_trim_aux:w ##1 \s__iow_mark }
    \cs_new:Npn \__iow_wrap_trim_aux:w ##1 \s__iow_mark ##2 \s__iow_stop {##1}
  }
\exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl
```

(*End of definition for* `\__iow_wrap_trim:N`*,* `\__iow_wrap_trim:w`*, and* `\__iow_wrap_trim_aux:w`*.*)

```
⟨@@=file⟩
```

50.4 File operations

`\l__file_internal_tl`

Used as a short-term scratch variable.

```
\tl_new:N \l__file_internal_tl
```

(*End of definition for* `\l__file_internal_tl`*.*)

\g_file_curr_dir_str
\g_file_curr_ext_str
\g_file_curr_name_str

The name of the current file should be available at all times: the name itself is set dynamically.

```
\str_new:N \g_file_curr_dir_str
\str_new:N \g_file_curr_ext_str
\str_new:N \g_file_curr_name_str
```

(*End of definition for* `\g_file_curr_dir_str`*,* `\g_file_curr_ext_str`*, and* `\g_file_curr_name_str`*. These variables are documented on page* *100*.)

\g__file_stack_seq

The input list of files is stored as a sequence stack. In package mode we can recover the information from the details held by LaTeX 2ε (we must be in the preamble and loaded using `\usepackage` or `\RequirePackage`). As LaTeX 2ε doesn't store directory and name separately, we stick to the same convention here. In pre-loading, `\@currnamestack` is empty so is skipped.

```
\seq_new:N \g__file_stack_seq
\group_begin:
  \cs_set_protected:Npn \__file_tmp:w #1#2#3
    {
      \tl_if_blank:nTF {#1}
        {
          \cs_set:Npn \__file_tmp:w ##1 " ##2 " ##3 \s__file_stop
            { { } {##2} {  } }
          \seq_gput_right:Ne \g__file_stack_seq
            {
              \exp_after:wN \__file_tmp:w \tex_jobname:D
                " \tex_jobname:D " \s__file_stop
            }
        }
        {
          \seq_gput_right:Nn \g__file_stack_seq { { } {#1} {#2} }
          \__file_tmp:w
        }
    }
  \cs_if_exist:NT \@currnamestack
    {
      \tl_if_empty:NF \@currnamestack
        { \exp_after:wN \__file_tmp:w \@currnamestack }
    }
\group_end:
```

(*End of definition for* `\g__file_stack_seq`.)

\g__file_record_seq

The total list of files used is recorded separately from the current file stack, as nothing is ever popped from this list. The current file name should be included in the file list! We will eventually copy the contents of `\@filelist`.

```
\seq_new:N \g__file_record_seq
```

(*End of definition for* `\g__file_record_seq`.)

\l__file_base_name_tl
\l__file_full_name_tl

For storing the basename and full path whilst passing data internally.

```
\tl_new:N \l__file_base_name_tl
\tl_new:N \l__file_full_name_tl
```

(*End of definition for* `\l__file_base_name_tl` *and* `\l__file_full_name_tl`.)

\l__file_dir_str
\l__file_ext_str
\l__file_name_str

Used in parsing a path into parts: in contrast to the above, these are never used outside of the current module.

```
\str_new:N \l__file_dir_str
\str_new:N \l__file_ext_str
\str_new:N \l__file_name_str
```

(*End of definition for* `\l__file_dir_str`*,* `\l__file_ext_str`*, and* `\l__file_name_str`*.*)

\l_file_search_path_seq

The current search path.

```
\seq_new:N \l_file_search_path_seq
```

(*End of definition for* `\l_file_search_path_seq`*. This variable is documented on page 101.*)

\l__file_tmp_seq

Scratch space for comma list conversion.

```
\seq_new:N \l__file_tmp_seq
```

(*End of definition for* `\l__file_tmp_seq`*.*)

50.4.1 Internal auxiliaries

\s__file_stop

Internal scan marks.

```
\scan_new:N \s__file_stop
```

(*End of definition for* `\s__file_stop`*.*)

\q__file_nil

Internal quarks.

```
\quark_new:N \q__file_nil
```

(*End of definition for* `\q__file_nil`*.*)

__file_quark_if_nil_p:n
__file_quark_if_nil:nTF

Branching quark conditional.

```
\__kernel_quark_new_conditional:Nn \__file_quark_if_nil:n { TF }
```

(*End of definition for* `\__file_quark_if_nil:nTF`*.*)

\q__file_recursion_tail
\q__file_recursion_stop

Internal recursion quarks.

```
\quark_new:N \q__file_recursion_tail
\quark_new:N \q__file_recursion_stop
```

(*End of definition for* `\q__file_recursion_tail` *and* `\q__file_recursion_stop`*.*)

__file_if_recursion_tail_break:NN
__file_if_recursion_tail_stop_do:Nn

Functions to query recursion quarks.

```
\__kernel_quark_new_test:N \__file_if_recursion_tail_stop:N
\__kernel_quark_new_test:N \__file_if_recursion_tail_stop_do:nn
```

(*End of definition for* `\__file_if_recursion_tail_break:NN` *and* `\__file_if_recursion_tail_stop_do:Nn`*.*)

__kernel_file_name_sanitize:n
__file_name_expand:n
__file_name_expand_cleanup:Nw
__file_name_expand_cleanup:w
__file_name_expand_end:
__file_name_expand_error:Nw
__file_name_expand_error_aux:Nw
__file_name_strip_quotes:n
__file_name_strip_quotes:nnnw
__file_name_strip_quotes:nnn
__file_name_trim_spaces:n
__file_name_trim_spaces:nw
__file_name_trim_spaces_aux:n
__file_name_trim_spaces_aux:w

Expanding the file name uses a `\csname`-based approach, and relies on active characters (for example from UTF-8 characters) being properly set up to expand to a expansion-safe version using `\ifcsname`. This is less conservative than the token-by-token approach used before, but it is much faster.

```
\cs_new:Npn \__kernel_file_name_sanitize:n #1
  {
    \exp_args:Ne \__file_name_trim_spaces:n
      {
        \exp_args:Ne \__file_name_strip_quotes:n
          { \__file_name_expand:n {#1} }
      }
  }
```

We'll use `\cs:w` to start expanding the file name, and to avoid creating csnames equal to `\relax` with "common" names, there's a prefix `__file_name=` to the csname. There's also a guard token at the end so we can check if there was an error during the process and (try to) clean up gracefully.

```
\cs_new:Npn \__file_name_expand:n #1
  {
    \exp_after:wN \__file_name_expand_cleanup:Nw
      \cs:w __file_name = #1 \cs_end:
        \__file_name_expand_end:
  }
```

With the csname built, we grab it, and grab the remaining tokens delimited by `\__file_name_expand_end:`. If there are any remaining tokens, something bad happened, so we'll call the error procedure `\__file_name_expand_error:Nw`. If everything went according to plan, then use `\token_to_str:N` on the csname built, and call `\__file_name_expand_cleanup:w` to remove the prefix we added a while back. `\__file_name_expand_cleanup:w` takes a leading argument so we don't have to bother about the value of `\tex_escapechar:D`.

```
\cs_new:Npn \__file_name_expand_cleanup:Nw #1 #2 \__file_name_expand_end:
  {
    \tl_if_empty:nF {#2}
      { \__file_name_expand_error:Nw #2 \__file_name_expand_end: }
    \exp_after:wN \__file_name_expand_cleanup:w \token_to_str:N #1
  }
\exp_last_unbraced:NNNNo
\cs_new:Npn \__file_name_expand_cleanup:w #1 \tl_to_str:n { __file_name = } { }
```

In non-error cases `\__file_name_expand_end:` should not expand. It will only do so in case there is a `\csname` too much in the file name, so it will throw an error (while expanding), then insert the missing `\cs_end:` and yet another `\__file_name_expand_end:` that will be used as a delimiter by `\__file_name_expand_cleanup:Nw` (or that will expand again if yet another `\endcsname` is missing).

```
\cs_new:Npn \__file_name_expand_end:
  {
    \msg_expandable_error:nn
      { kernel } { filename-missing-endcsname }
    \cs_end: \__file_name_expand_end:
  }
```

Now to the error case. `\__file_name_expand_error:Nw` adds an extra `\cs_end:` so that in case there was an extra `\csname` in the file name, then `\__file_name_expand_error_aux:Nw` throws the error.

```
\cs_new:Npn \__file_name_expand_error:Nw #1 #2 \__file_name_expand_end:
  { \__file_name_expand_error_aux:Nw #1 #2 \cs_end: \__file_name_expand_end: }
\cs_new:Npn \__file_name_expand_error_aux:Nw #1 #2 \cs_end: #3
    \__file_name_expand_end:
  {
    \msg_expandable_error:nnff
      { kernel } { filename-chars-lost }
        { \token_to_str:N #1 } { \exp_stop_f: #2 }
  }
```

Quoting file name uses basically the same approach as for `luaquotejobname`: count the `"` tokens and remove them.

```
\cs_new:Npn \__file_name_strip_quotes:n #1
  {
    \__file_name_strip_quotes:nw { 0 }
      #1 " \q__file_recursion_tail " \q__file_recursion_stop {#1}
  }
\cs_new:Npn \__file_name_strip_quotes:nw #1#2 "
  {
    \if_meaning:w \q__file_recursion_tail #2
      \__file_name_strip_quotes_end:wnwn
    \fi:
    #2
    \__file_name_strip_quotes:nw { #1 + 1 }
  }
\cs_new:Npn \__file_name_strip_quotes_end:wnwn \fi: #1
    \__file_name_strip_quotes:nw #2 \q__file_recursion_stop #3
  {
    \fi:
    \int_if_odd:nT {#2}
      {
        \msg_expandable_error:nnn
          { kernel } { unbalanced-quote-in-filename } {#3}
      }
  }
```

Spaces need to be trimmed from the start of the name and from the end of any extension. However, the name we are passed might not have an extension: that means we have to look for one. If there is no extension, we still use the standard trimming function but deliberately prevent any spaces being removed at the end.

```
\cs_new:Npn \__file_name_trim_spaces:n #1
  { \__file_name_trim_spaces:nw {#1} #1 . \q__file_nil . \s__file_stop }
\cs_new:Npn \__file_name_trim_spaces:nw #1#2 . #3 . #4 \s__file_stop
  {
    \__file_quark_if_nil:nTF {#3}
      {
        \tl_trim_spaces_apply:nN { #1 \s__file_stop }
          \__file_name_trim_spaces_aux:n
      }
      { \tl_trim_spaces:n {#1} }
  }
```

```
\cs_new:Npn \__file_name_trim_spaces_aux:n #1
  { \__file_name_trim_spaces_aux:w #1 }
\cs_new:Npn \__file_name_trim_spaces_aux:w #1 \s__file_stop {#1}
```

(*End of definition for* `\__kernel_file_name_sanitize:n` *and others.*)

`\__kernel_file_name_quote:n`
`\__file_name_quote:nw`

```
\cs_new:Npn \__kernel_file_name_quote:n #1
  { \__file_name_quote:nw {#1} #1 ~ \q__file_nil \s__file_stop }
\cs_new:Npn \__file_name_quote:nw #1 #2 ~ #3 \s__file_stop
  {
    \__file_quark_if_nil:nTF {#3}
      { #1 }
      { "#1" }
  }
```

(*End of definition for* `\__kernel_file_name_quote:n` *and* `\__file_name_quote:nw`*.*)

`\c__file_marker_tl`

The same idea as the marker for rescanning token lists: this pair of tokens cannot appear in a file that is being input.

```
\tl_const:Ne \c__file_marker_tl { : \token_to_str:N : }
```

(*End of definition for* `\c__file_marker_tl`*.*)

`\file_get:nnNTF`
`\file_get:VnNTF`
`\file_get:nnN`
`\__file_get_aux:nnN`
`\__file_get_do:Nw`

The approach here is similar to that for `\tl_set_rescan:Nnn`. The file contents are grabbed as an argument delimited by `\c__file_marker_tl`. A few subtleties: braces in `\if_false:` ... `\fi:` to deal with possible alignment tabs, `\tracingnesting` to avoid a warning about a group being closed inside the `\scantokens`, and `\prg_return_true:` is placed after the end-of-file marker.

```
\cs_new_protected:Npn \file_get:nnN #1#2#3
  {
    \file_get:nnNF {#1} {#2} #3
      { \tl_set:Nn #3 { \q_no_value } }
  }
\cs_generate_variant:Nn \file_get:nnN { V }
\prg_new_protected_conditional:Npnn \file_get:nnN #1#2#3 { T , F , TF }
  {
    \file_get_full_name:nNTF {#1} \l__file_full_name_tl
      {
        \exp_args:NV \__file_get_aux:nnN
          \l__file_full_name_tl
          {#2} #3
        \prg_return_true:
      }
      { \prg_return_false: }
  }
\prg_generate_conditional_variant:Nnn \file_get:nnN { V } { T , F , TF }
\cs_new_protected:Npe \__file_get_aux:nnN #1#2#3
  {
    \exp_not:N \if_false: { \exp_not:N \fi:
    \group_begin:
      \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
      \exp_not:N \exp_args:No \tex_everyeof:D
        { \exp_not:N \c__file_marker_tl }
```

```
      #2 \scan_stop:
      \exp_not:N \exp_after:wN \exp_not:N \__file_get_do:Nw
      \exp_not:N \exp_after:wN #3
      \exp_not:N \exp_after:wN \exp_not:N \prg_do_nothing:
      \exp_not:N \tex_input:D
      \sys_if_engine_luatex:TF
        { {#1} }
        { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
    \exp_not:N \if_false: } \exp_not:N \fi:
  }
\exp_args:Nno \use:nn
  { \cs_new_protected:Npn \__file_get_do:Nw #1#2 }
  { \c__file_marker_tl }
  {
    \group_end:
    \tl_set:No #1 {#2}
  }
```

(*End of definition for* `\file_get:nnNTF` *and others. These functions are documented on page 104.*)

`\__file_size:n`

A copy of the primitive where it's available.

```
\cs_new_eq:NN \__file_size:n \tex_filesize:D
```

(*End of definition for* `\__file_size:n`*.*)

`\file_full_name:n`
`\__file_full_name:n`
`\__file_full_name_aux:n`
`\__file_full_name_auxi:nn`
`\__file_full_name_auxii:nn`
`\__file_full_name_aux:Nnn`
`\__file_full_name_slash:n`
`\__file_full_name_slash:w`
`\__file_full_name_aux:nN`
`\__file_full_name_aux:nnN`
`\__file_name_cleanup:w`
`\__file_name_end:`
`\__file_name_ext_check:nn`
`\__file_name_ext_check:nnw`
`\__file_name_ext_check:nnnw`
`\__file_name_ext_check:nnn`
`\__file_name_ext_check:nnnn`

File searching can be carried out if the `\pdffilesize` primitive or an equivalent is available. That of course means we need to arrange for everything else to here to be done by expansion too. We start off by sanitizing the name and quoting if required: we may need to remove those quotes, so the raw name is passed too.

```
\cs_new:Npn \file_full_name:n #1
  {
    \exp_args:Ne \__file_full_name:n
      { \__kernel_file_name_sanitize:n {#1} }
  }
\cs_generate_variant:Nn \file_full_name:n { V }
```

First, we check of the file is just here: no mapping so we do not need the break part of the broader auxiliary. We are using the fact that the primitive here returns nothing if the file is entirely absent. To avoid unnecessary filesystem lookups, the result of `\pdffilesize` is kept available as an argument. For package mode, `\input@path` is a token list not a sequence.

```
\cs_new:Npn \__file_full_name:n #1
  {
    \tl_if_blank:nF {#1}
      { \exp_args:Nne \__file_full_name_auxii:nn {#1} { \__file_full_name_aux:n {#1} } }
  }
```

To avoid repeated reading of files we need to cache the loading: this is important as the code here is used by *all* file checks. The same marker is used in the $\LaTeX2_\varepsilon$ kernel, meaning that we get a double-saving with for example `\IfFileExists`. As this is all about performance, we use the low-level approach for the conditionals. For a file already seen, the size is reported as -1 so it's distinct from any non-cached ones.

```
\cs_new:Npn \__file_full_name_aux:n #1
  {
```

```
    \if_cs_exist:w __file_seen_ \tl_to_str:n {#1} : \cs_end:
      -1
    \else:
      \exp_args:Ne \__file_full_name_auxi:nn { \__file_size:n {#1} } {#1}
    \fi:
  }
```

We will need the size of files later, and we have to avoid the `\scan_stop:` causing issues if we are raising the flag. Thus there is a slightly odd gobble here.

```
\cs_new:Npn \__file_full_name_auxi:nn #1#2
  {
    \if:w \scan_stop: #1 \scan_stop:
    \else:
      \exp_after:wN \use_none:n
        \cs:w __file_seen_ \tl_to_str:n {#2} : \cs_end:
      #1
    \fi:
  }
\cs_new:Npn \__file_full_name_auxii:nn #1 #2
  {
    \tl_if_blank:nTF {#2}
      {
        \seq_map_tokens:Nn \l_file_search_path_seq
          { \__file_full_name_aux:Nnn \seq_map_break:n {#1} }
        \cs_if_exist:NT \input@path
          {
            \tl_map_tokens:Nn \input@path
              { \__file_full_name_aux:Nnn \tl_map_break:n {#1} }
          }
        \__file_name_end:
      }
      { \__file_ext_check:nn {#1} {#2} }
  }
```

Two pars to the auxiliary here so we can avoid doing quoting twice in the event we find the right file.

```
\cs_new:Npn \__file_full_name_aux:Nnn #1#2#3
  {
    \exp_args:Ne \__file_full_name_aux:nN
      { \__file_full_name_slash:n {#3} #2 }
      #1
  }
\cs_new:Npn \__file_full_name_slash:n #1
  {
    \__file_full_name_slash:nw {#1} #1 \q_nil / \q_nil / \q_nil \q_stop
  }
\cs_new:Npn \__file_full_name_slash:nw #1#2 / \q_nil / #3 \q_stop
  {
    \quark_if_nil:nTF {#3}
      { #1 / }
      { #2 / }
  }
\cs_new:Npn \__file_full_name_aux:nN #1
  { \exp_args:Nne \__file_full_name_aux:nnN {#1} { \__file_full_name_aux:n {#1} } }
\cs_new:Npn \__file_full_name_aux:nnN #1 #2 #3
```

```
  {
    \tl_if_blank:nF {#2}
      {
        #3
          {
            \__file_ext_check:nn {#1} {#2}
            \__file_name_cleanup:w
          }
      }
  }
\cs_new:Npn \__file_name_cleanup:w #1 \__file_name_end: { }
\cs_new:Npn \__file_name_end: { }
```

As TeX automatically adds .tex if there is no extension, there is a little clean up to do here. First, make sure we are not in the directory part, saving that. Then check for an extension.

```
\cs_new:Npn \__file_ext_check:nn #1 #2
{ \__file_ext_check:nnw {#2} { / } #1 / \q__file_nil / \s__file_stop }
\cs_new:Npn \__file_ext_check:nnw #1 #2 #3 / #4 / #5 \s__file_stop
  {
    \__file_quark_if_nil:nTF {#4}
      {
        \exp_args:No \__file_ext_check:nnnw
          { \use_none:n #2 } {#1} {#3} #3 . \q__file_nil . \s__file_stop
      }
      { \__file_ext_check:nnw {#1} { #2 #3 / } #4 / #5 \s__file_stop }
  }
\cs_new:Npe \__file_ext_check:nnnw #1#2#3#4 . #5 . #6 \s__file_stop
  {
    \exp_not:N \__file_quark_if_nil:nTF {#5}
      {
        \exp_not:N \__file_ext_check:nnn
          { #1 #3 \tl_to_str:n { .tex } } { #1 #3 } {#2}
      }
      { #1 #3 }
  }
\cs_new:Npn \__file_ext_check:nnn #1
  { \exp_args:Nne \__file_ext_check:nnnn {#1} { \__file_full_name_aux:n {#1} } }
\cs_new:Npn \__file_ext_check:nnnn #1#2#3#4
  {
    \tl_if_blank:nTF {#2}
      {#3}
      {
        \bool_lazy_or:nnTF
          { \int_compare_p:nNn {#4} = {#2} }
          { \int_compare_p:nNn {#2} = { -1 } }
          {#1}
          {#3}
      }
  }
```

(End of definition for `\file_full_name:n` *and others. This function is documented on page* *103.)*

`\file_get_full_name:nN`
`\file_get_full_name:VN`
`\file_get_full_name:nNTF`
`\file_get_full_name:VNTF`
`\__file_get_full_name_search:nN`

These functions pre-date using `\tex_filesize:D` for file searching, so are `get` functions with protection. To avoid having different search set ups, they are simply wrappers around the code above.

```
\cs_new_protected:Npn \file_get_full_name:nN #1#2
  {
    \file_get_full_name:nNF {#1} #2
      { \tl_set:Nn #2 { \q_no_value } }
  }
\cs_generate_variant:Nn \file_get_full_name:nN { V }
\prg_new_protected_conditional:Npnn \file_get_full_name:nN #1#2 { T , F , TF }
  {
    \__kernel_tl_set:Nx #2
      { \file_full_name:n {#1} }
    \tl_if_empty:NTF #2
      { \prg_return_false: }
      { \prg_return_true: }
  }
\prg_generate_conditional_variant:Nnn \file_get_full_name:nN
  { V } { T , F ,  TF }
```

(*End of definition for* `\file_get_full_name:nN`*,* `\file_get_full_name:nNTF`*, and* `\__file_get_full_name_search:nN`*. These functions are documented on page 103.*)

`\g__file_internal_ior`

A reserved stream to test for opening a shell.

```
\ior_new:N \g__file_internal_ior
```

(*End of definition for* `\g__file_internal_ior`*.*)

`\file_mdfive_hash:n`
`\file_mdfive_hash:V`
`\file_size:n`
`\file_size:V`
`\file_timestamp:n`
`\file_timestamp:V`
`\__file_details:nn`
`\__file_details_aux:nn`
`\__file_mdfive_hash:n`

Getting file details by expansion is relatively easy if a bit repetitive. As the MD5 function has a slightly different syntax from the other commands, there is a little cleaning up to do.

```
\cs_new:Npn \file_size:n #1
  { \__file_details:nn {#1} { size } }
\cs_generate_variant:Nn \file_size:n { V }
\cs_new:Npn \file_timestamp:n #1
  { \__file_details:nn {#1} { moddate } }
\cs_generate_variant:Nn \file_timestamp:n { V }
\cs_new:Npn \__file_details:nn #1#2
  {
    \exp_args:Ne \__file_details_aux:nn
      { \file_full_name:n {#1} } {#2}
  }
\cs_new:Npn \__file_details_aux:nn #1#2
  {
    \tl_if_blank:nF {#1}
      { \use:c { tex_file #2 :D } {#1} }
  }
\cs_new:Npn \file_mdfive_hash:n #1
  { \exp_args:Ne \__file_mdfive_hash:n { \file_full_name:n {#1} } }
\cs_generate_variant:Nn \file_mdfive_hash:n { V }
\cs_new:Npn \__file_mdfive_hash:n #1
  { \tex_mdfivesum:D file {#1} }
```

(*End of definition for* `\file_mdfive_hash:n` *and others. These functions are documented on page 102.*)

\file_hex_dump:nnn
\file_hex_dump:Vnn
__file_hex_dump_auxi:nnn
__file_hex_dump_auxii:nnnn
__file_hex_dump_auxiii:nnnn
__file_hex_dump_auxiv:nnn
\file_hex_dump:n
\file_hex_dump:V
__file_hex_dump:n

These are separate as they need multiple arguments *or* the file size. For LuaTeX, the emulation does not need the file size so we save a little on expansion.

```
\cs_new:Npn \file_hex_dump:nnn #1#2#3
  {
    \exp_args:Neee \__file_hex_dump_auxi:nnn
      { \file_full_name:n {#1} }
      { \int_eval:n {#2} }
      { \int_eval:n {#3} }
  }
\cs_generate_variant:Nn \file_hex_dump:nnn { V }
\cs_new:Npn \__file_hex_dump_auxi:nnn #1#2#3
  {
    \bool_lazy_any:nF
      {
        { \tl_if_blank_p:n {#1} }
        { \int_compare_p:nNn {#2} = 0 }
        { \int_compare_p:nNn {#3} = 0 }
      }
      {
        \exp_args:Ne \__file_hex_dump_auxii:nnnn
          { \__file_details_aux:nn {#1} { size } }
          {#1} {#2} {#3}
      }
  }
\cs_new:Npn \__file_hex_dump_auxii:nnnn #1#2#3#4
  {
    \int_compare:nNnTF {#3} > 0
      { \__file_hex_dump_auxiii:nnnn {#3} }
      {
        \exp_args:Ne \__file_hex_dump_auxiii:nnnn
          { \int_eval:n { #1 + #3 } }
      }
        {#1} {#2} {#4}
  }
\cs_new:Npn \__file_hex_dump_auxiii:nnnn #1#2#3#4
  {
    \int_compare:nNnTF {#4} > 0
      { \__file_hex_dump_auxiv:nnn {#4} }
      {
        \exp_args:Ne \__file_hex_dump_auxiv:nnn
          { \int_eval:n { #2 + #4 } }
      }
        {#1} {#3}
  }
\cs_new:Npn \__file_hex_dump_auxiv:nnn #1#2#3
  {
    \tex_filedump:D
      offset ~ \int_eval:n { #2 - 1 } ~
      length ~ \int_eval:n { #1 - #2 + 1 }
      {#3}
  }
\cs_new:Npn \file_hex_dump:n #1
  { \exp_args:Ne \__file_hex_dump:n { \file_full_name:n {#1} } }
\cs_generate_variant:Nn \file_hex_dump:n { V }
```

```
\sys_if_engine_luatex:TF
  {
    \cs_new:Npn \__file_hex_dump:n #1
      {
        \tl_if_blank:nF {#1}
          { \tex_filedump:D whole {#1} {#1} }
      }
  }
  {
    \cs_new:Npn \__file_hex_dump:n #1
      {
        \tl_if_blank:nF {#1}
          { \tex_filedump:D length \tex_filesize:D {#1} {#1} }
      }
  }
```

(*End of definition for* `\file_hex_dump:nnn` *and others. These functions are documented on page 101.*)

`\file_get_hex_dump:nN`
`\file_get_hex_dump:VN`
`\file_get_hex_dump:nNTF`
`\file_get_hex_dump:VNTF`
`\file_get_mdfive_hash:nN`
`\file_get_mdfive_hash:VN`
`\file_get_mdfive_hash:nNTF`
`\file_get_mdfive_hash:VNTF`
`\file_get_size:nN`
`\file_get_size:VN`
`\file_get_size:nNTF`
`\file_get_size:VNTF`
`\file_get_timestamp:nN`
`\file_get_timestamp:VN`
`\file_get_timestamp:nNTF`
`\file_get_timestamp:VNTF`
`\__file_get_details:nnN`

Non-expandable wrappers around the above in the case where appropriate primitive support exists.

```
\cs_new_protected:Npn \file_get_hex_dump:nN #1#2
  { \file_get_hex_dump:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_generate_variant:Nn \file_get_hex_dump:nN { V }
\cs_new_protected:Npn \file_get_mdfive_hash:nN #1#2
  { \file_get_mdfive_hash:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_generate_variant:Nn \file_get_mdfive_hash:nN { V }
\cs_new_protected:Npn \file_get_size:nN #1#2
  { \file_get_size:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_generate_variant:Nn \file_get_size:nN { V }
\cs_new_protected:Npn \file_get_timestamp:nN #1#2
  { \file_get_timestamp:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
\cs_generate_variant:Nn \file_get_timestamp:nN { V }
\prg_new_protected_conditional:Npnn \file_get_hex_dump:nN #1#2 { T , F , TF }
  { \__file_get_details:nnN {#1} { hex_dump } #2 }
\prg_generate_conditional_variant:Nnn \file_get_hex_dump:nN
  { V } { T , F , TF }
\prg_new_protected_conditional:Npnn \file_get_mdfive_hash:nN #1#2 { T , F , TF }
  { \__file_get_details:nnN {#1} { mdfive_hash } #2 }
\prg_generate_conditional_variant:Nnn \file_get_mdfive_hash:nN
  { V } { T , F , TF }
\prg_new_protected_conditional:Npnn \file_get_size:nN #1#2 { T , F , TF }
  { \__file_get_details:nnN {#1} { size } #2 }
\prg_generate_conditional_variant:Nnn \file_get_size:nN
  { V } { T , F , TF }
\prg_new_protected_conditional:Npnn \file_get_timestamp:nN #1#2 { T , F , TF }
  { \__file_get_details:nnN {#1} { timestamp } #2 }
\prg_generate_conditional_variant:Nnn \file_get_timestamp:nN
  { V } { T , F , TF }
\cs_new_protected:Npn \__file_get_details:nnN #1#2#3
  {
    \__kernel_tl_set:Nx #3
      { \use:c { file_ #2 :n } {#1} }
    \tl_if_empty:NTF #3
      { \prg_return_false: }
```

```
      { \prg_return_true: }
  }
```

(*End of definition for* `\file_get_hex_dump:nNTF` *and others. These functions are documented on page* 101.)

`\file_get_hex_dump:nnnN` `\file_get_hex_dump:VnnN` `\file_get_hex_dump:nnnNTF` `\file_get_hex_dump:VnnNTF`

Custom code due to the additional arguments.

```
\cs_new_protected:Npn \file_get_hex_dump:nnnN #1#2#3#4
  {
    \file_get_hex_dump:nnnNF {#1} {#2} {#3} #4
      { \tl_set:Nn #4 { \q_no_value } }
  }
\cs_generate_variant:Nn \file_get_hex_dump:nnnN { V }
\prg_new_protected_conditional:Npnn \file_get_hex_dump:nnnN #1#2#3#4
  { T , F , TF }
  {
    \__kernel_tl_set:Nx #4
      { \file_hex_dump:nnn {#1} {#2} {#3} }
    \tl_if_empty:NTF #4
      { \prg_return_false: }
      { \prg_return_true: }
  }
\prg_generate_conditional_variant:Nnn \file_get_hex_dump:nnnN
  { V } { T , F , TF }
```

(*End of definition for* `\file_get_hex_dump:nnnNTF`*. This function is documented on page* 101.)

`\__file_str_cmp:nn`

As we are doing a fixed-length "big" integer comparison, it is easiest to use the low-level behavior of string comparisons.

```
\cs_new_eq:NN \__file_str_cmp:nn \tex_strcmp:D
```

(*End of definition for* `\__file_str_cmp:nn`*.*)

`\file_compare_timestamp_p:nNn` `\file_compare_timestamp_p:nNV` `\file_compare_timestamp_p:VNn` `\file_compare_timestamp_p:VNV` `\file_compare_timestamp:nNnTF` `\file_compare_timestamp:nNVTF` `\file_compare_timestamp:VNnTF` `\file_compare_timestamp:VNVTF` `\__file_compare_timestamp:nnN` `\__file_timestamp:n`

Comparison of file date can be done by using the low-level nature of the string comparison functions.

```
\prg_new_conditional:Npnn \file_compare_timestamp:nNn #1#2#3
  { p , T , F , TF }
  {
    \exp_args:Nee \__file_compare_timestamp:nnN
      { \file_full_name:n {#1} }
      { \file_full_name:n {#3} }
      #2
  }
\prg_generate_conditional_variant:Nnn \file_compare_timestamp:nNn
  { nNV , V , VNV } { p , T , F , TF }
\cs_new:Npn \__file_compare_timestamp:nnN #1#2#3
  {
    \tl_if_blank:nTF {#1}
      {
        \if_charcode:w #3 <
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      }
```

```
        {
          \tl_if_blank:nTF {#2}
            {
              \if_charcode:w #3 >
                \prg_return_true:
              \else:
                \prg_return_false:
              \fi:
            }
            {
              \if_int_compare:w
                \__file_str_cmp:nn
                  { \__file_timestamp:n {#1} }
                  { \__file_timestamp:n {#2} }
                  #3 \c_zero_int
                \prg_return_true:
              \else:
                \prg_return_false:
              \fi:
            }
        }
    }
\cs_new_eq:NN \__file_timestamp:n \tex_filemoddate:D
```

(*End of definition for* `\file_compare_timestamp:nNnTF`, `\__file_compare_timestamp:nnN`, *and* `\__file_timestamp:n`. *This function is documented on page 103.*)

`\file_if_exist_p:n`
`\file_if_exist_p:V`
`\file_if_exist:nTF`
`\file_if_exist:VTF`

The test for the existence of a file is a wrapper around the function to add a path to a file. If the file was found, the path contains something, whereas if the file was not located then the return value is empty.

```
\prg_new_conditional:Npnn \file_if_exist:n #1 { p , T , F , TF }
  {
    \tl_if_blank:eTF { \file_full_name:n {#1} }
      { \prg_return_false: }
      { \prg_return_true: }
  }
\prg_generate_conditional_variant:Nnn \file_if_exist:n { V } { p , T , F , TF }
```

(*End of definition for* `\file_if_exist:nTF`. *This function is documented on page 101.*)

`\file_if_exist_input:n`
`\file_if_exist_input:V`
`\file_if_exist_input:nF`
`\file_if_exist_input:VF`

Input of a file with a test for existence. We do not define the `T` or `TF` variants because the most useful place to place the ⟨*true code*⟩ would be inconsistent with other conditionals.

```
\cs_new_protected:Npn \file_if_exist_input:n #1
  {
    \file_get_full_name:nNT {#1} \l__file_full_name_tl
      { \__file_input:V \l__file_full_name_tl }
  }
\cs_generate_variant:Nn \file_if_exist_input:n { V }
\cs_new_protected:Npn \file_if_exist_input:nF #1#2
  {
    \file_get_full_name:nNTF {#1} \l__file_full_name_tl
      { \__file_input:V \l__file_full_name_tl }
      {#2}
  }
\cs_generate_variant:Nn \file_if_exist_input:nF { V }
```

(*End of definition for* `\file_if_exist_input:n` *and* `\file_if_exist_input:nF`. *These functions are documented on page 104.*)

`\file_input_stop:` A simple rename.

```
\cs_new_protected:Npn \file_input_stop: { \tex_endinput:D }
```

(*End of definition for* `\file_input_stop:`. *This function is documented on page 105.*)

`\__kernel_file_missing:n` An error message for a missing file, also used in `\ior_open:Nn`.

```
\cs_new_protected:Npn \__kernel_file_missing:n #1
  {
    \msg_error:nne { kernel } { file-not-found }
      { \__kernel_file_name_sanitize:n {#1} }
  }
```

(*End of definition for* `\__kernel_file_missing:n`.)

`\file_input:n`
`\file_input:V`
`\__file_input:n`
`\__file_input:V`
`\__file_input_push:n`
`\__kernel_file_input_push:n`
`\__file_input_pop:`
`\__kernel_file_input_pop:`
`\__file_input_pop:nnn`

Loading a file is done in a safe way, checking first that the file exists and loading only if it does. Push the file name on the `\g__file_stack_seq`, and add it to the file list, either `\g__file_record_seq`, or `\@filelist` in package mode.

```
\cs_new_protected:Npn \file_input:n #1
  {
    \file_get_full_name:nNTF {#1} \l__file_full_name_tl
      { \__file_input:V \l__file_full_name_tl }
      { \__kernel_file_missing:n {#1} }
  }
\cs_generate_variant:Nn \file_input:n { V }
\cs_new_protected:Npe \__file_input:n #1
  {
    \exp_not:N \clist_if_exist:NTF \exp_not:N \@filelist
      { \exp_not:N \@addtofilelist {#1} }
      { \seq_gput_right:Nn \exp_not:N \g__file_record_seq {#1} }
    \exp_not:N \__file_input_push:n {#1}
    \exp_not:N \tex_input:D
    \sys_if_engine_luatex:TF
      { {#1} }
      { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
    \exp_not:N \__file_input_pop:
  }
\cs_generate_variant:Nn \__file_input:n { V }
```

Keeping a track of the file data is easy enough: we store the separated parts so we do not need to parse them twice.

```
\cs_new_protected:Npn \__file_input_push:n #1
  {
    \seq_gpush:Ne \g__file_stack_seq
      {
        { \g_file_curr_dir_str }
        { \g_file_curr_name_str }
        { \g_file_curr_ext_str }
      }
    \file_parse_full_name:nNNN {#1}
      \l__file_dir_str \l__file_name_str \l__file_ext_str
    \str_gset_eq:NN \g_file_curr_dir_str  \l__file_dir_str
    \str_gset_eq:NN \g_file_curr_name_str \l__file_name_str
```

```
    \str_gset_eq:NN \g_file_curr_ext_str  \l__file_ext_str
  }
\cs_new_eq:NN \__kernel_file_input_push:n \__file_input_push:n
\cs_new_protected:Npn \__file_input_pop:
  {
    \seq_gpop:NN \g__file_stack_seq \l__file_internal_tl
    \exp_after:wN \__file_input_pop:nnn \l__file_internal_tl
  }
\cs_new_eq:NN \__kernel_file_input_pop: \__file_input_pop:
\cs_new_protected:Npn \__file_input_pop:nnn #1#2#3
  {
    \str_gset:Nn \g_file_curr_dir_str  {#1}
    \str_gset:Nn \g_file_curr_name_str {#2}
    \str_gset:Nn \g_file_curr_ext_str  {#3}
  }
```

(*End of definition for* `\file_input:n` *and others. This function is documented on page* *104*.)

`\file_input_raw:n`
`\file_input_raw:V`
`\__file_input_raw:nn`

No error checking, no tracking.

```
\cs_new:Npn \file_input_raw:n #1
  { \exp_args:Ne \__file_input_raw:nn { \file_full_name:n {#1} } {#1} }
\cs_generate_variant:Nn \file_input_raw:n { V }
\cs_new:Npe \__file_input_raw:nn #1#2
  {
    \exp_not:N \tl_if_blank:nTF {#1}
      {
        \exp_not:N \exp_args:Nnne \exp_not:N \msg_expandable_error:nnn
          { kernel } { file-not-found }
          { \exp_not:N \__kernel_file_name_sanitize:n {#2} }
      }
      {
        \exp_not:N \tex_input:D
          \sys_if_engine_luatex:TF
            { {#1} }
            { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
        }
  }
\exp_args_generate:n { nne }
```

(*End of definition for* `\file_input_raw:n` *and* `\__file_input_raw:nn`*. This function is documented on page* *104*.)

`\file_parse_full_name:n`
`\file_parse_full_name:V`
`\file_parse_full_name_apply:nN`
`\file_parse_full_name_apply:VN`

The main parsing macro `\file_parse_full_name_apply:nN` passes the file name `#1` through `\__kernel_file_name_sanitize:n` so that we have a single normalised way to treat files internally. `\file_parse_full_name:n` uses the former, with `\prg_do_nothing:` to leave each part of the name within a pair of braces.

```
\cs_new:Npn \file_parse_full_name:n #1
  {
    \file_parse_full_name_apply:nN {#1}
      \prg_do_nothing:
  }
\cs_generate_variant:Nn \file_parse_full_name:n { V }
\cs_new:Npn \file_parse_full_name_apply:nN #1
  {
```

```
    \exp_args:Ne \__file_parse_full_name_auxi:nN
      { \__kernel_file_name_sanitize:n {#1} }
  }
\cs_generate_variant:Nn \file_parse_full_name_apply:nN { V }
```

`\__file_parse_full_name_area:nw` splits the file name into chunks separated by `/`, until the last one is reached. The last chunk is the file name plus the extension, and everything before that is the path. When `\__file_parse_full_name_area:nw` is done, it leaves the path within braces after the scan mark `\s__file_stop` and proceeds parsing the actual file name.

__file_parse_full_name_auxi:nN
__file_parse_full_name_area:nw

```
\cs_new:Npn \__file_parse_full_name_auxi:nN #1
  {
    \__file_parse_full_name_area:nw { } #1
      / \s__file_stop
  }
\cs_new:Npn \__file_parse_full_name_area:nw #1 #2 / #3 \s__file_stop
  {
    \tl_if_empty:nTF {#3}
      { \__file_parse_full_name_base:nw { } #2 . \s__file_stop {#1} }
      { \__file_parse_full_name_area:nw { #1 / #2 } #3 \s__file_stop }
  }
```

`\__file_parse_full_name_base:nw` does roughly the same as above, but it separates the chunks at each period. However here there's some extra complications: In case `#1` is empty, it is assumed that the extension is actually empty, and the file name is `#2`. Besides, an extra `.` has to be added to `#2` because it is later removed in `\__file_parse_full_name_tidy:nnnN`. In any case, if there's an extension, it is returned with a leading `.`.

__file_parse_full_name_base:nw

```
\cs_new:Npn \__file_parse_full_name_base:nw #1 #2 . #3 \s__file_stop
  {
    \tl_if_empty:nTF {#3}
      {
        \tl_if_empty:nTF {#1}
          {
            \tl_if_empty:nTF {#2}
              { \__file_parse_full_name_tidy:nnnN { } { } }
              { \__file_parse_full_name_tidy:nnnN { .#2 } { } }
          }
          { \__file_parse_full_name_tidy:nnnN {#1} { .#2 } }
      }
      { \__file_parse_full_name_base:nw { #1 . #2 } #3 \s__file_stop }
  }
```

Now we just need to tidy some bits left loose before. The loop used in the two macros above start with a leading `/` and `.` in the file path an name, so here we need to remove them, except in the path, if it is a single `/`, in which case it's left as is. After all's done, pass to `#4`.

__file_parse_full_name_tidy:nnnN

```
\cs_new:Npn \__file_parse_full_name_tidy:nnnN #1 #2 #3 #4
  {
    \exp_args:Nee #4
      {
        \str_if_eq:nnF {#3} { / } { \use_none:n }
        #3 \prg_do_nothing:
```

```
    }
    { \use_none:n #1 \prg_do_nothing: }
    {#2}
  }
```

(*End of definition for* `\file_parse_full_name:n` *and others. These functions are documented on page* 104.)

`\file_parse_full_name:nNNN`
`\file_parse_full_name:VNNN`

```
\cs_new_protected:Npn \file_parse_full_name:nNNN #1 #2 #3 #4
  {
    \file_parse_full_name_apply:nN {#1}
      \__file_full_name_assign:nnnNNN #2 #3 #4
  }
\cs_new_protected:Npn \__file_full_name_assign:nnnNNN #1 #2 #3 #4 #5 #6
  {
    \str_set:Nn #4 {#1}
    \str_set:Nn #5 {#2}
    \str_set:Nn #6 {#3}
  }
\cs_generate_variant:Nn \file_parse_full_name:nNNN { V }
```

(*End of definition for* `\file_parse_full_name:nNNN`*. This function is documented on page* 103.)

`\file_show_list:`
`\file_log_list:`
`\__file_list:N`
`\__file_list_aux:n`

A function to list all files used to the log, without duplicates. In package mode, if `\@filelist` is still defined, we need to take this list of file names into account (we capture it `\AtBeginDocument` into `\g__file_record_seq`), turning it to a string (this does not affect the commas of this comma list).

```
\cs_new_protected:Npn \file_show_list: { \__file_list:N \msg_show:nneeee }
\cs_new_protected:Npn \file_log_list: { \__file_list:N \msg_log:nneeee }
\cs_new_protected:Npn \__file_list:N #1
  {
    \seq_clear:N \l__file_tmp_seq
    \clist_if_exist:NT \@filelist
      {
        \exp_args:NNe \seq_set_from_clist:Nn \l__file_tmp_seq
          { \tl_to_str:N \@filelist }
      }
    \seq_concat:NNN \l__file_tmp_seq \l__file_tmp_seq \g__file_record_seq
    \seq_remove_duplicates:N \l__file_tmp_seq
    #1 { kernel } { file-list }
      { \seq_map_function:NN \l__file_tmp_seq \__file_list_aux:n }
        { } { } { }
  }
\cs_new:Npn \__file_list_aux:n #1 { \iow_newline: #1 }
```

(*End of definition for* `\file_show_list:` *and others. These functions are documented on page* 105.)

When used as a package, there is a need to hold onto the standard file list as well as the new one here. File names recorded in `\@filelist` must be turned to strings before being added to `\g__file_record_seq`.

```
\cs_if_exist:NT \@filelist
  {
    \AtBeginDocument
      {
```

```
        \exp_args:NNe \seq_set_from_clist:Nn \l__file_tmp_seq
          { \tl_to_str:N \@filelist }
        \seq_gconcat:NNN
          \g__file_record_seq
          \g__file_record_seq
          \l__file_tmp_seq
      }
  }
```

50.5 GetIdInfo

\GetIdInfo
__file_id_info_auxi:w
__file_id_info_auxii:w
__file_id_info_auxiii:w

As documented in expl3.dtx this function extracts file name etc from an SVN `Id` line. This used to be how we got version number and so on in all modules, so it had to be defined in l3bootstrap. Now it's more convenient to define it after we have set up quite a lot of tools, and l3file seems the least unreasonable place for it.

The idea here is to extract out the information needed from a standard SVN `Id` line, but to avoid a line that would get changed when the file is checked in. Hence the fact that none of the lines here include both a dollar sign and the `Id` keyword!

```
\cs_new_protected:Npn \GetIdInfo
  {
    \tl_clear_new:N \ExplFileDescription
    \tl_clear_new:N \ExplFileDate
    \tl_clear_new:N \ExplFileName
    \tl_clear_new:N \ExplFileExtension
    \tl_clear_new:N \ExplFileVersion
    \group_begin:
    \char_set_catcode_space:n { 32 }
    \exp_after:wN
    \group_end:
    \__file_id_info_auxi:w
  }
```

A first check for a completely empty SVN field. If that is not the case, there is a second case when a file created using `svn cp` but has not been checked in. That leaves a special marker `-1` version, which has no further data. Dealing correctly with that is the reason for the space in the line to use `\__file_id_info_auxii:w`.

```
\cs_new_protected:Npn \__file_id_info_auxi:w $ #1 $ #2
  {
    \tl_set:Nn \ExplFileDescription {#2}
    \str_if_eq:nnTF {#1} { Id }
      {
        \tl_set:Nn \ExplFileDate { 0000/00/00 }
        \tl_set:Nn \ExplFileName { [unknown] }
        \tl_set:Nn \ExplFileExtension { [unknown~extension] }
        \tl_set:Nn \ExplFileVersion {-1}
      }
      { \__file_id_info_auxii:w #1 ~ \s__file_stop }
  }
```

Here, `#1` is `Id`, `#2` is the file name, `#3` is the extension, `#4` is the version, `#5` is the check in date and `#6` is the check in time and user, plus some trailing spaces. If `#4` is the marker `-1` value then `#5` and `#6` are empty.

```
\cs_new_protected:Npn \__file_id_info_auxii:w
```

```
    #1 ~ #2.#3 ~ #4 ~ #5 ~ #6 \s__file_stop
  {
    \tl_set:Nn \ExplFileName {#2}
    \tl_set:Nn \ExplFileExtension {#3}
    \tl_set:Nn \ExplFileVersion {#4}
    \str_if_eq:nnTF {#4} {-1}
      { \tl_set:Nn \ExplFileDate { 0000/00/00 } }
      { \__file_id_info_auxiii:w #5 - 0 - 0 - \s__file_stop }
  }
```

Convert an SVN-style date into a LaTeX-style one.

```
\cs_new_protected:Npn \__file_id_info_auxiii:w #1 - #2 - #3 - #4 \s__file_stop
  { \tl_set:Nn \ExplFileDate { #1/#2/#3 } }
```

(*End of definition for* `\GetIdInfo` *and others. This function is documented on page* 11.)

50.6 Checking the version of kernel dependencies

__kernel_dependency_version_check:Nn
__kernel_dependency_version_check:nn
__file_kernel_dependency_compare:nnn
__file_parse_version:w

This function is responsible for checking if dependencies of the LaTeX3 kernel match the version preloaded in the LaTeX 2_ε kernel. If versions don't match, the function attempts to tell why by searching for a possible stray format file.

The function starts by checking that the kernel date is defined, and if not zero is used to force the error route. The kernel date is then compared with the argument requested date (usually the packaging date of the dependency). If the kernel date is less than the required date, it's an error and the loading should abort.

```
\cs_new_protected:Npn \__kernel_dependency_version_check:Nn #1
  { \exp_args:NV \__kernel_dependency_version_check:nn #1 }
\cs_new_protected:Npn \__kernel_dependency_version_check:nn #1
  {
    \cs_if_exist:NTF \c__kernel_expl_date_tl
      {
        \exp_args:NV \__file_kernel_dependency_compare:nnn
          \c__kernel_expl_date_tl {#1}
      }
      { \__file_kernel_dependency_compare:nnn { 0000-00-00 } {#1} }
  }
\cs_new_protected:Npn \__file_kernel_dependency_compare:nnn #1 #2 #3
  {
    \int_compare:nNnT
        { \__file_parse_version:w #1 \s__file_stop } <
        { \__file_parse_version:w #2 \s__file_stop }
      { \__file_mismatched_dependency_error:nn {#2} {#3} }
  }
\cs_new:Npn \__file_parse_version:w #1 - #2 - #3 \s__file_stop {#1#2#3}
```

If the versions differ, then we try to give the user some guidance. This function starts by taking the engine name `\c_sys_engine_str` and replacing `tex` by `latex`, then building a command of the form: kpsewhich –all –engine=⟨*engine*⟩ ⟨*format*⟩[-dev].fmt to query the format files available. A shell is opened and each line is read into a sequence.

__file_mismatched_dependency_error:nn

```
\cs_new_protected:Npn \__file_mismatched_dependency_error:nn #1 #2
  {
    \exp_args:NNe \ior_shell_open:Nn \g__file_internal_ior
      {
```

```
        kpsewhich ~ --all ~
          --engine = \c_sys_engine_exec_str
          \c_space_tl \c_sys_engine_format_str
            \bool_lazy_and:nnT
                { \tl_if_exist_p:N \development@branch@name }
                { ! \tl_if_empty_p:N \development@branch@name }
              { -dev } .fmt
      }
    \seq_clear:N \l__file_tmp_seq
    \ior_map_inline:Nn \g__file_internal_ior
      { \seq_put_right:Nn \l__file_tmp_seq {##1} }
    \ior_close:N \g__file_internal_ior
    \msg_error:nnnn { kernel } { mismatched-support-file }
      {#1} {#2}
```

And finish by ending the current file.

```
    \tex_endinput:D
  }
```

Now define the actual error message:

```
\msg_new:nnnn { kernel } { mismatched-support-file }
  {
    Mismatched~LaTeX~support~files~detected. \\
    Loading~'#2'~aborted!
```

`\c__kernel_expl_date_tl` may not exist, due to an older format, so only print the dates when the sentinel token list exists:

```
    \tl_if_exist:NT \c__kernel_expl_date_tl
      {
        \\ \\
        The~L3~programming~layer~in~the~LaTeX~format \\
        is~dated~\c__kernel_expl_date_tl,~but~in~your~TeX~
        tree~the~files~require \\ at~least~#1.
      }
  }
  {
```

The sequence containing the format files should have exactly one item: the format file currently being run. If that's the case, the cause of the error is not that, so print a generic help with some possible causes. If more than one format file was found, then print the list to the user, with appropriate indications of what's in the system and what's in the user tree.

```
    \int_compare:nNnTF { \seq_count:N \l__file_tmp_seq } > 1
      {
        The~cause~seems~to~be~an~old~format~file~in~the~user~tree. \\
        LaTeX~found~these~files:
        \seq_map_tokens:Nn \l__file_tmp_seq { \\~-~\use:n } \\
        Try~deleting~the~file~in~the~user~tree~then~run~LaTeX~again.
      }
      {
        The~most~likely~causes~are:
        \\~-~A~recent~format~generation~failed;
        \\~-~A~stray~format~file~in~the~user~tree~which~needs~
            to~be~removed~or~rebuilt;
        \\~-~You~are~running~a~manually~installed~version~of~#2 \\
```

```
      \ \ \ which~is~incompatible~with~the~version~in~LaTeX. \\
    }
  \\
  LaTeX~will~abort~loading~the~incompatible~support~files~
  but~this~may~lead~to \\ later~errors.~Please~ensure~that~
  your~LaTeX~format~is~correctly~regenerated.
}
```

(*End of definition for* `\__kernel_dependency_version_check:Nn` *and others.*)

50.7 Messages

```
\msg_new:nnnn { kernel } { file-not-found }
  { File~'#1'~not~found. }
  {
    The~requested~file~could~not~be~found~in~the~current~directory,~
    in~the~TeX~search~path~or~in~the~LaTeX~search~path.
  }
\msg_new:nnn { kernel } { file-list }
  {
    >~File~List~<
    #1 \\
    .............
  }
\msg_new:nnnn { kernel } { filename-chars-lost }
  { #1~invalid~in~file~name.~Lost:~#2. }
  {
    There~was~an~invalid~token~in~the~file~name~that~caused~
    the~characters~following~it~to~be~lost.
  }
\msg_new:nnnn { kernel } { filename-missing-endcsname }
  { Missing~\iow_char:N\\endcsname~inserted~in~filename. }
  {
    The~file~name~had~more~\iow_char:N\\csname~commands~than~
    \iow_char:N\\endcsname~ones.~LaTeX~will~add~the~missing~
    \iow_char:N\\endcsname~and~try~to~continue~as~best~as~it~can.
  }
\msg_new:nnnn { kernel } { unbalanced-quote-in-filename }
  { Unbalanced~quotes~in~file~name~'#1'. }
  {
    File~names~must~contain~balanced~numbers~of~quotes~(").
  }
\msg_new:nnnn { kernel } { iow-indent }
  { Only~#1 allows~#2 }
  {
    The~command~#2 can~only~be~used~in~messages~
    which~will~be~wrapped~using~#1.
    \tl_if_empty:nF {#3} { ~ It~was~called~with~argument~'#3'. }
  }
```

50.8 Functions delayed from earlier modules

<@@=sys>

`\c_sys_platform_str` Detecting the platform on LuaTeX is easy: for other engines, we use the fact that the two common cases have special null files. It is possible to probe further (see package platform), but that requires shell escape and seems unlikely to be useful. This is set up here as it requires file searching.

```
\sys_if_engine_luatex:TF
  {
    \str_const:Ne \c_sys_platform_str
      { \tex_directlua:D { tex.print(os.type) } }
  }
  {
    \file_if_exist:nTF { nul: }
      {
        \file_if_exist:nF { /dev/null }
          { \str_const:Nn \c_sys_platform_str { windows } }
      }
      {
        \file_if_exist:nT { /dev/null }
          { \str_const:Nn \c_sys_platform_str { unix } }
      }
  }
\cs_if_exist:NF \c_sys_platform_str
  { \str_const:Nn \c_sys_platform_str { unknown }  }
```

(*End of definition for* `\c_sys_platform_str`. *This variable is documented on page 77.*)

`\sys_if_platform_unix_p:`
`\sys_if_platform_unix:TF`
`\sys_if_platform_windows_p:`
`\sys_if_platform_windows:TF`

We can now set up the tests.

```
\clist_map_inline:nn { unix , windows }
  {
    \__file_const:nn { sys_if_platform_ #1 }
      { \str_if_eq_p:Vn \c_sys_platform_str { #1 } }
  }
```

(*End of definition for* `\sys_if_platform_unix:TF` *and* `\sys_if_platform_windows:TF`. *These functions are documented on page 77.*)

```
⟨/package⟩
```

Chapter 51

l3luatex implementation

```
⟨*package⟩
```

51.1 Breaking out to Lua

```
⟨*tex⟩
⟨@@=lua⟩
```

__lua_escape:n
__lua_now:n
__lua_shipout:n

Copies of primitives.

```
\cs_new_eq:NN \__lua_escape:n  \tex_luaescapestring:D
\cs_new_eq:NN \__lua_now:n     \tex_directlua:D
\cs_new_eq:NN \__lua_shipout:n \tex_latelua:D
```

(*End of definition for* `\__lua_escape:n`*,* `\__lua_now:n`*, and* `\__lua_shipout:n`*.*)

These functions are set up in l3str for bootstrapping: we want to replace them with a "proper" version at this stage, so clean up.

```
\cs_undefine:N \lua_escape:e
\cs_undefine:N \lua_now:e
```

\lua_now:n
\lua_now:e
\lua_shipout_e:n
\lua_shipout:n
\lua_escape:n
\lua_escape:e

Wrappers around the primitives.

```
\cs_new:Npn \lua_now:e #1 { \__lua_now:n {#1} }
\cs_new:Npn \lua_now:n #1 { \lua_now:e { \exp_not:n {#1} } }
\cs_new_protected:Npn \lua_shipout_e:n #1 { \__lua_shipout:n {#1} }
\cs_new_protected:Npn \lua_shipout:n #1
  { \lua_shipout_e:n { \exp_not:n {#1} } }
\cs_new:Npn \lua_escape:e #1 { \__lua_escape:n {#1} }
\cs_new:Npn \lua_escape:n #1 { \lua_escape:e { \exp_not:n {#1} } }
```

(*End of definition for* `\lua_now:n` *and others. These functions are documented on page 106.*)

\lua_load_module:n

Wrapper around `require'`⟨*module*⟩`'`.

```
\str_new:N \l__lua_err_msg_str
\cs_new_protected:Npn \lua_load_module:n #1
  {
    \bool_if:nF { \__lua_load_module_p:n { #1 } }
      {
        \msg_error:nnnV
          { luatex } { module-not-found } { #1 } \l__lua_err_msg_str
      }
  }
```

(*End of definition for* `\lua_load_module:n`. *This function is documented on page 107.*)

As with engines other than LuaTeX these have to be macros, we give them the same status in all cases. When LuaTeX is not in use, simply give an error message/

```
\sys_if_engine_luatex:F
  {
    \clist_map_inline:nn
      {
        \lua_escape:n , \lua_escape:e ,
        \lua_now:n , \lua_now:e
      }
      {
        \cs_gset:Npn #1 ##1
          {
            \msg_expandable_error:nnn
              { luatex } { luatex-required } { #1 }
          }
      }
    \clist_map_inline:nn
      { \lua_shipout_e:n , \lua_shipout:n, \lua_load_module:n }
      {
        \cs_gset_protected:Npn #1 ##1
          {
            \msg_error:nnn
              { luatex } { luatex-required } { #1 }
          }
      }
  }
```

51.2 Messages

```
\msg_new:nnnn { luatex } { luatex-required }
  { LuaTeX~engine~not~in~use!~Ignoring~#1. }
  {
    The~feature~you~are~using~is~only~available~
    with~the~LuaTeX~engine.~LaTeX3~ignored~'#1'.
  }

\msg_new:nnnn { luatex } { module-not-found }
  { Lua~module~`#1'~not~found. }
  {
    The~file~`#1.lua'~could~not~be~found.~Please~ensure~
    that~the~file~was~properly~installed~and~that~the~
    filename~database~is~current. \\ \\
    The~Lua~loader~provided~this~additional~information: \\
    #2
  }

\prop_gput:Nnn \g_msg_module_name_prop { luatex } { LaTeX }
\prop_gput:Nnn \g_msg_module_type_prop { luatex } { }
```

⟨/tex⟩

51.3 Lua functions for internal use

⟨*lua⟩

Most of the emulation of pdfTEX here is based heavily on Heiko Oberdiek's pdftexcmds package.

`ltx.utils` Create a table for the kernel's own use.

```
ltx = ltx or {utils={}}
ltx.utils = ltx.utils or { }
local ltxutils = ltx.utils
```

(*End of definition for* `ltx.utils`. *This function is documented on page 107.*)

Local copies of global tables.

```
local io       = io
local kpse     = kpse
local lfs      = lfs
local math     = math
local md5      = md5
local os       = os
local string   = string
local tex      = tex
local texio    = texio
local tonumber = tonumber
```

Local copies of standard functions.

```
local abs        = math.abs
local byte       = string.byte
local floor      = math.floor
local format     = string.format
local gsub       = string.gsub
local lfs_attr   = lfs.attributes
local open       = io.open
local os_date    = os.date
local setcatcode = tex.setcatcode
local sprint     = tex.sprint
local cprint     = tex.cprint
local write      = tex.write
local write_nl   = texio.write_nl
local utf8_char  = utf8.char
local package_loaded    = package.loaded
local package_searchers = package.searchers
local table_concat      = table.concat

local scan_int     = token.scan_int or token.scan_integer
local scan_string  = token.scan_string
local scan_keyword = token.scan_keyword
local put_next     = token.put_next
local token_create = token.create
local token_new    = token.new
local set_macro    = token.set_macro
```

Since token.create only returns useful values after the tokens has been added to TeX's hash table, we define a variant which defines it first if necessary.

```
local token_create_safe
do
  local is_defined = token.is_defined
  local set_char   = token.set_char
```

```
  local runtoks    = tex.runtoks
  local let_token  = token_create'let'

  function token_create_safe(s)
    local orig_token = token_create(s)
    if is_defined(s, true) then
      return orig_token
    end
    set_char(s, 0)
    local new_token = token_create(s)
    runtoks(function()
      put_next(let_token, new_token, orig_token)
    end)
    return new_token
  end
end

local true_tok     = token_create_safe'prg_return_true:'
local false_tok    = token_create_safe'prg_return_false:'
```

In ConTEXt lmtx `token.command_id` does not exist, but it can easily be emulated with ConTEXt's `tokens.commands`.

```
local command_id   = token.command_id
if not command_id and tokens and tokens.commands then
  local id_map = tokens.commands
  function command_id(name)
    return id_map[name]
  end
end
```

Deal with ConTEXt: doesn't use `kpse` library.

```
local kpse_find = (resolvers and resolvers.findfile) or kpse.find_file
```

`escapehex` An internal auxiliary to convert a string to the matching hex escape. This works on a byte basis: extension to handled UTF-8 input is covered in pdftexcmds but is not currently required here.

```
local function escapehex(str)
  return (gsub(str, ".",
    function (ch) return format("%02X", byte(ch)) end))
end
```

(*End of definition for* `escapehex`.)

`ltx.utils.filedump` Similar comments here to the next function: read the file in binary mode to avoid any line-end weirdness.

```
local function filedump(name,offset,length)
  local file = kpse_find(name,"tex",true)
  if not file then return end
  local f = open(file,"rb")
  if not f then return end
  if offset and offset > 0 then
    f:seek("set", offset)
  end
  local data = f:read(length or 'a')
  f:close()
```

```
  return escapehex(data)
end
ltxutils.filedump = filedump
```

(*End of definition for* `ltx.utils.filedump`*. This function is documented on page 107.*)

`md5.HEX` Hash a string and return the hash in uppercase hexadecimal format. In some engines, this is built-in. For traditional LuaTEX, the conversion to hexadecimal has to be done by us.

```
local md5_HEX = md5.HEX
if not md5_HEX then
  local md5_sum = md5.sum
  function md5_HEX(data)
    return escapehex(md5_sum(data))
  end
  md5.HEX = md5_HEX
end
```

(*End of definition for* `md5.HEX`*.*)

`ltx.utils.filemd5sum` Read an entire file and hash it: the hash function itself is a built-in. As Lua is byte-based there is no work needed here in terms of UTF-8 (see pdftexcmds and how it handles strings that have passed through LuaTEX). The file is read in binary mode so that no line ending normalisation occurs.

```
local function filemd5sum(name)
  local file = kpse_find(name, "tex", true) if not file then return end
  local f = open(file, "rb") if not f then return end

  local data = f:read("*a")
  f:close()
  return md5_HEX(data)
end
ltxutils.filemd5sum = filemd5sum
```

(*End of definition for* `ltx.utils.filemd5sum`*. This function is documented on page 107.*)

`ltx.utils.filemoddate` There are two cases: If the C standard library is C99 compliant, we can use `%z` to get the timezone in almost the right format. We only have to add primes and replace a zero or missing offset with `Z`.

Of course this would be boring, so Windows does things differently. There we have to manually calculate the offset. See procedure `makepdftime` in `utils.c` of pdfTEX.

```
local filemoddate
if os_date'%z':match'^[+-]%d%d%d%d$' then
  local pattern = lpeg.Cs(16 *
      (lpeg.Cg(lpeg.S'+-' * '0000' * lpeg.Cc'Z')
    + 3 * lpeg.Cc"'" * 2 * lpeg.Cc"'"
    + lpeg.Cc'Z')
  * -1)
  function filemoddate(name)
    local file = kpse_find(name, "tex", true)
    if not file then return end
    local date = lfs_attr(file, "modification")
    if not date then return end
    return pattern:match(os_date("D:%Y%m%d%H%M%S%z", date))
```

```
  end
else
  local function filemoddate(name)
    local file = kpse_find(name, "tex", true)
    if not file then return end
    local date = lfs_attr(file, "modification")
    if not date then return end
    local d = os_date("*t", date)
    local u = os_date("!*t", date)
    local off = 60 * (d.hour - u.hour) + d.min - u.min
    if d.year ~= u.year then
      if d.year > u.year then
        off = off + 1440
      else
        off = off - 1440
      end
    elseif d.yday ~= u.yday then
      if d.yday > u.yday then
        off = off + 1440
      else
        off = off - 1440
      end
    end
    local timezone
    if off == 0 then
      timezone = "Z"
    else
      if off < 0 then
        timezone = "-"
        off = -off
      else
        timezone = "+"
      end
      timezone = format("%s%02d'%02d'", timezone, hours // 60, hours % 60)
    end
    return format("D:%04d%02d%02d%02d%02d%02d%s",
        d.year, d.month, d.day, d.hour, d.min, d.sec, timezone)
  end
end
ltxutils.filemoddate = filemoddate
```

(*End of definition for* `ltx.utils.filemoddate`*. This function is documented on page* *107*.)

`ltx.utils.filesize` A simple disk lookup.

```
local function filesize(name)
  local file = kpse_find(name, "tex", true)
  if file then
    local size = lfs_attr(file, "size")
    if size then
      return size
    end
  end
end
ltxutils.filesize = filesize
```

(*End of definition for* `ltx.utils.filesize`. *This function is documented on page* 108.)

`luacmd` An internal function for defining control sequences form Lua which behave like primitives. This acts as a wrapper around `token.set_lua` which accepts a function instead of an index into the functions table.

```
local luacmd do
  local set_lua = token.set_lua
  local undefined_cs = command_id'undefined_cs'

  if not context and not luatexbase then require'ltluatex' end
  if luatexbase then
    local new_luafunction = luatexbase.new_luafunction
    local functions = lua.get_functions_table()
    function luacmd(name, func, ...)
      local id
      local tok = token_create(name)
      if tok.command == undefined_cs then
        id = new_luafunction(name)
        set_lua(name, id, ...)
      else
        id = tok.index or tok.mode
      end
      functions[id] = func
    end
  elseif context then
    local register = context.functions.register
    local functions = context.functions.known
    function luacmd(name, func, ...)
      local tok = token_create(name)
      if tok.command == undefined_cs then
        token.set_lua(name, register(func), ...)
      else
        functions[tok.index or tok.mode] = func
      end
    end
  end
end
```

(*End of definition for* `luacmd`.)

`try_require` Loads a Lua module. This function loads the module similarly to the standard Lua global function `require`, with a few differences. On success, `try_require` returns `true, module`. If the module cannot be found, it returns `false, err_msg`. If the module is found, but something goes wrong when loading it, the function throws an error.

```
local function try_require(name)
  if package_loaded[name] then
    return true, package_loaded[name]
  end

  local failure_details = {}
  for _, searcher in ipairs(package_searchers) do
    local loader, data = searcher(name)
    if type(loader) == 'function' then
      package_loaded[name] = loader(name, data) or true
```

```
      return true, package_loaded[name]
    elseif type(loader) == 'string' then
      failure_details[#failure_details + 1] = loader
    end
  end

  return false, table_concat(failure_details, '\n')
end
```

(*End of definition for* `try_require`.)

`\__lua_load_module_p:n` Check to see if we can load a module using `require`. If we can load the module, then we load it immediately. Otherwise, we save the error message in `\l_@@_err_msg_str`.

```
local char_given   = command_id'char_given'
local c_true_bool  = token_create(1, char_given)
local c_false_bool = token_create(0, char_given)
local c_str_cctab  = token_create('c_str_cctab').mode

luacmd('__lua_load_module_p:n', function()
  local success, result = try_require(scan_string())
  if success then
    set_macro(c_str_cctab, 'l__lua_err_msg_str', '')
    put_next(c_true_bool)
  else
    set_macro(c_str_cctab, 'l__lua_err_msg_str', result)
    put_next(c_false_bool)
  end
end)
```

(*End of definition for* `\__lua_load_module_p:n`.)

51.4 Preserving iniTeX Lua data for runs

```
⟨@@=lua⟩
```

The Lua state is not dumped when a format is written, therefore any Lua variables filled doing format building need to be restored in order to be accessible during normal runs.

We provide some kernel-internal helpers for this. They will only be available if `luatexbase` is available. This is not a big restriction though, because ConTEXt (which does not use `luatexbase`) does not load expl3 in the format.

```
local register_luadata, get_luadata

if luatexbase then
  local register = token_create'@expl@luadata@bytecode'.index
  if status.ini_version then
```

`register_luadata` `register_luadata` is only available during format generation. It accept a string which uniquely identifies the data object and has to be provided to retrieve it later. Additionally it accepts a function which is called in the `pre_dump` callback and which has to return a string that evaluates to a valid Lua object to be preserved.

```
    local luadata, luadata_order = {}, {}

    function register_luadata(name, func)
```

```
      if luadata[name] then
        error(format("LaTeX error: data name %q already in use", name))
      end
      luadata[name] = func
      luadata_order[#luadata_order + 1] = func and name
    end
```

(*End of definition for* `register_luadata`.)

The actual work is done in `pre_dump`. The `luadata_order` is used to ensure that the order is consistent over multiple runs.

```
    luatexbase.add_to_callback("pre_dump", function()
      if next(luadata) then
        local str = "return {"
        for i=1, #luadata_order do
          local name = luadata_order[i]
          str = format('%s[%q]=%s,', str, name, luadata[name]())
        end
        lua.bytecode[register] = assert(load(str .. "}"))
      end
    end, "ltx.luadata")
  else
```

get_luadata

`get_luadata` is only available if data should be restored. It accept the identifier which was used when the data object was registered and returns the associated object. Every object can only be retrieved once.

```
    local luadata = lua.bytecode[register]
    if luadata then
      lua.bytecode[register] = nil
      luadata = luadata()
    end
    function get_luadata(name)
      if not luadata then return end
      local data = luadata[name]
      luadata[name] = nil
      return data
    end
  end
end
```

(*End of definition for* `get_luadata`.)

⟨/lua⟩

⟨/package⟩

Chapter 52

l3legacy implementation

```
⟨*package⟩
⟨@@=legacy⟩
```

\legacy_if_p:n
\legacy_if:nTF

A friendly wrapper. We need to use the `\if:w` approach here, rather than testing against `\iftrue`/`\iffalse` as the latter approach fails for primitive conditionals such as `\ifmmode`. The `\reverse_if:N` here means that we get a slightly more useful error if the name is undefined.

```
\prg_new_conditional:Npnn \legacy_if:n #1 { p , T , F , TF }
  {
    \exp_after:wN \reverse_if:N
      \cs:w if#1 \cs_end:
      \prg_return_false:
    \else:
      \prg_return_true:
    \fi:
  }
```

(*End of definition for* `\legacy_if:nTF`. *This function is documented on page 109.*)

\legacy_if_set_true:n
\legacy_if_set_false:n
\legacy_if_gset_true:n
\legacy_if_gset_false:n

A friendly wrapper.

```
\cs_new_protected:Npn \legacy_if_set_true:n #1
  { \cs_set_eq:cN { if#1 } \if_true: }
\cs_new_protected:Npn \legacy_if_set_false:n #1
  { \cs_set_eq:cN { if#1 } \if_false: }
\cs_new_protected:Npn \legacy_if_gset_true:n #1
  { \cs_gset_eq:cN { if#1 } \if_true: }
\cs_new_protected:Npn \legacy_if_gset_false:n #1
  { \cs_gset_eq:cN { if#1 } \if_false: }
```

(*End of definition for* `\legacy_if_set_true:n` *and others. These functions are documented on page 109.*)

\legacy_if_set:nn
\legacy_if_gset:nn

A more elaborate wrapper.

```
\cs_new_protected:Npn \legacy_if_set:nn #1#2
  {
    \bool_if:nTF {#2} \legacy_if_set_true:n \legacy_if_set_false:n
    {#1}
  }
```

```
\cs_new_protected:Npn \legacy_if_gset:nn #1#2
  {
    \bool_if:nTF {#2} \legacy_if_gset_true:n \legacy_if_gset_false:n
    {#1}
  }
```

(*End of definition for* `\legacy_if_set:nn` *and* `\legacy_if_gset:nn`*. These functions are documented on page 109.*)

```
⟨/package⟩
```

Chapter 53

l3tl implementation

```
⟨*package⟩
⟨@@=tl⟩
```

A token list variable is a TeX macro that holds tokens. By using the ε-TeX primitive `\unexpanded` inside a TeX `\edef` it is possible to store any tokens, including `#`, in this way.

53.1 Functions

`\__kernel_tl_set:Nx`
`\__kernel_tl_gset:Nx`

These two are supplied to get better performance for macros which would otherwise use `\tl_set:Ne` or `\tl_gset:Ne` internally.

```
\cs_new_eq:NN \__kernel_tl_set:Nx  \cs_set_nopar:Npe
\cs_new_eq:NN \__kernel_tl_gset:Nx \cs_gset_nopar:Npe
```

(*End of definition for* `\__kernel_tl_set:Nx` *and* `\__kernel_tl_gset:Nx`*.*)

`\tl_new:N`
`\tl_new:c`

Creating new token list variables is a case of checking for an existing definition and doing the definition.

```
\cs_new_protected:Npn \tl_new:N #1
  {
    \__kernel_chk_if_free_cs:N #1
    \cs_gset_eq:NN #1 \c_empty_tl
  }
\cs_generate_variant:Nn \tl_new:N { c }
```

(*End of definition for* `\tl_new:N`*. This function is documented on page 111.*)

`\tl_const:Nn`
`\tl_const:Ne`
`\tl_const:Nx`
`\tl_const:cn`
`\tl_const:ce`
`\tl_const:cx`

Constants are also easy to generate. They use `\cs_gset_nopar:Npe` instead of `\__kernel_tl_gset:Nx` so that the correct scope checking for `c`, instead of for `g`, is applied when `\debug_on:n { check-declarations }` is used. Constant assignment functions are patched specially in l3debug to apply such checks.

```
\cs_new_protected:Npn \tl_const:Nn #1#2
  {
    \__kernel_chk_if_free_cs:N #1
    \cs_gset_nopar:Npe #1 { \__kernel_exp_not:w {#2} }
  }
\cs_generate_variant:Nn \tl_const:Nn { Ne , c , ce }
\cs_generate_variant:Nn \tl_const:Nn { Nx , cx }
```

(*End of definition for* `\tl_const:Nn`. *This function is documented on page 112.*)

`\tl_clear:N`
`\tl_clear:c`
`\tl_gclear:N`
`\tl_gclear:c`

Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```
\cs_new_protected:Npn \tl_clear:N  #1
  { \tex_let:D #1 = ~ \c_empty_tl }
\cs_new_protected:Npn \tl_gclear:N #1
  { \tex_global:D \tex_let:D #1 ~ \c_empty_tl }
\cs_generate_variant:Nn \tl_clear:N  { c }
\cs_generate_variant:Nn \tl_gclear:N { c }
```

(*End of definition for* `\tl_clear:N` *and* `\tl_gclear:N`. *These functions are documented on page 112.*)

`\tl_clear_new:N`
`\tl_clear_new:c`
`\tl_gclear_new:N`
`\tl_gclear_new:c`

Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```
\cs_new_protected:Npn \tl_clear_new:N  #1
  { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
\cs_new_protected:Npn \tl_gclear_new:N #1
  { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
\cs_generate_variant:Nn \tl_clear_new:N  { c }
\cs_generate_variant:Nn \tl_gclear_new:N { c }
```

(*End of definition for* `\tl_clear_new:N` *and* `\tl_gclear_new:N`. *These functions are documented on page 112.*)

`\tl_set_eq:NN`
`\tl_set_eq:Nc`
`\tl_set_eq:cN`
`\tl_set_eq:cc`
`\tl_gset_eq:NN`
`\tl_gset_eq:Nc`
`\tl_gset_eq:cN`
`\tl_gset_eq:cc`

For setting token list variables equal to each other. To allow for patching, the arguments have to be explicit. In addition this ensures that a braced second argument will not cause problems.

```
\cs_new_protected:Npn \tl_set_eq:NN  #1#2
  { \tex_let:D #1 = ~ #2 }
\cs_new_protected:Npn \tl_gset_eq:NN #1#2
  { \tex_global:D \tex_let:D #1 = ~ #2 }
\cs_generate_variant:Nn \tl_set_eq:NN { cN, Nc, cc }
\cs_generate_variant:Nn \tl_gset_eq:NN { cN, Nc, cc }
```

(*End of definition for* `\tl_set_eq:NN` *and* `\tl_gset_eq:NN`. *These functions are documented on page 112.*)

`\tl_concat:NNN`
`\tl_concat:ccc`
`\tl_gconcat:NNN`
`\tl_gconcat:ccc`

Concatenating token lists is easy. When checking is turned on, all three arguments must be checked: a token list `#2` or `#3` equal to `\scan_stop:` would lead to problems later on.

```
\cs_new_protected:Npn \tl_concat:NNN #1#2#3
  {
    \__kernel_tl_set:Nx #1
      {
        \__kernel_exp_not:w \exp_after:wN {#2}
        \__kernel_exp_not:w \exp_after:wN {#3}
      }
  }
\cs_new_protected:Npn \tl_gconcat:NNN #1#2#3
  {
    \__kernel_tl_gset:Nx #1
      {
        \__kernel_exp_not:w \exp_after:wN {#2}
        \__kernel_exp_not:w \exp_after:wN {#3}
```

```
      }
  }
\cs_generate_variant:Nn \tl_concat:NNN  { ccc }
\cs_generate_variant:Nn \tl_gconcat:NNN { ccc }
```

(*End of definition for* `\tl_concat:NNN` *and* `\tl_gconcat:NNN`*. These functions are documented on page 112.*)

`\tl_if_exist_p:N`
`\tl_if_exist_p:c`
`\tl_if_exist:NTF`
`\tl_if_exist:cTF`

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }
\prg_new_eq_conditional:NNn \tl_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(*End of definition for* `\tl_if_exist:NTF`*. This function is documented on page 112.*)

53.2 Constant token lists

`\c_empty_tl`

Never full. We need to define that constant before using `\tl_new:N`.

```
\tl_const:Nn \c_empty_tl { }
```

(*End of definition for* `\c_empty_tl`*. This variable is documented on page 126.*)

`\c_novalue_tl`

A special marker: as we don't have `\char_generate:nn` yet, has to be created the old-fashioned way.

```
\group_begin:
\tex_catcode:D `- = 11 ~
\tl_const:Ne \c_novalue_tl { - NoValue \token_to_str:N - }
\group_end:
```

(*End of definition for* `\c_novalue_tl`*. This variable is documented on page 127.*)

`\c_space_tl`

A space as a token list (as opposed to as a character).

```
\tl_const:Nn \c_space_tl { ~ }
```

(*End of definition for* `\c_space_tl`*. This variable is documented on page 127.*)

53.3 Adding to token list variables

`\tl_set:Nn`
`\tl_set:NV`
`\tl_set:Nv`
`\tl_set:No`
`\tl_set:Ne`
`\tl_set:Nf`
`\tl_set:Nx`
`\tl_set:cn`
`\tl_set:cV`
`\tl_set:cv`
`\tl_set:co`
`\tl_set:ce`
`\tl_set:cf`
`\tl_set:cx`
`\tl_gset:Nn`
`\tl_gset:NV`
`\tl_gset:Nv`
`\tl_gset:No`
`\tl_gset:Ne`
`\tl_gset:Nf`
`\tl_gset:Nx`
`\tl_gset:cn`
`\tl_gset:cV`
`\tl_gset:cv`
`\tl_gset:co`

By using `\exp_not:n` token list variables can contain `#` tokens, which makes the token list registers provided by TeX more or less redundant. The `\tl_set:No` version is done by hand as it is used quite a lot.

```
\cs_new_protected:Npn \tl_set:Nn #1#2
  { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w {#2} } }
\cs_new_protected:Npn \tl_set:No #1#2
  { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w \exp_after:wN {#2} } }
\cs_new_protected:Npn \tl_gset:Nn #1#2
  { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w {#2} } }
\cs_new_protected:Npn \tl_gset:No #1#2
  { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w \exp_after:wN {#2} } }
\cs_generate_variant:Nn \tl_set:Nn  {     NV , Nv , Ne , Nf }
\cs_generate_variant:Nn \tl_set:Nn  { c, cV , cv , ce , cf }
\cs_generate_variant:Nn \tl_set:No  { c }
\cs_generate_variant:Nn \tl_set:Nn  { Nx , cx }
\cs_generate_variant:Nn \tl_gset:Nn {     NV , Nv , Ne , Nf }
```

```
\cs_generate_variant:Nn \tl_gset:Nn { c, cV , cv , ce , cf }
\cs_generate_variant:Nn \tl_gset:No { c }
\cs_generate_variant:Nn \tl_gset:Nn { Nx , cx }
```

(*End of definition for* `\tl_set:Nn` *and* `\tl_gset:Nn`. *These functions are documented on page 112.*)

`\tl_put_left:Nn` `\tl_put_left:NV` `\tl_put_left:Nv` `\tl_put_left:Ne` `\tl_put_left:No` `\tl_put_left:Nx` `\tl_put_left:cn` `\tl_put_left:cV` `\tl_put_left:cv` `\tl_put_left:ce` `\tl_put_left:co` `\tl_put_left:cx` `\tl_gput_left:Nn` `\tl_gput_left:NV` `\tl_gput_left:Nv` `\tl_gput_left:Ne` `\tl_gput_left:No` `\tl_gput_left:Nx` `\tl_gput_left:cn` `\tl_gput_left:cV` `\tl_gput_left:cv` `\tl_gput_left:ce` `\tl_gput_left:co` `\tl_gput_left:cx`

Adding to the left is done directly to gain a little performance.

```
\cs_new_protected:Npn \tl_put_left:Nn #1#2
  {
    \__kernel_tl_set:Nx #1
      { \__kernel_exp_not:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
  }
\cs_new_protected:Npn \tl_put_left:NV #1#2
  {
    \__kernel_tl_set:Nx #1
      { \exp_not:V #2 \__kernel_exp_not:w \exp_after:wN {#1} }
  }
\cs_new_protected:Npn \tl_put_left:Nv #1#2
  {
    \__kernel_tl_set:Nx #1
      { \exp_not:v {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
  }
\cs_new_protected:Npn \tl_put_left:Ne #1#2
  {
    \__kernel_tl_set:Nx #1
      {
        \__kernel_exp_not:w \tex_expanded:D { {#2} }
        \__kernel_exp_not:w \exp_after:wN {#1}
      }
  }
\cs_new_protected:Npn \tl_put_left:No #1#2
  {
    \__kernel_tl_set:Nx #1
      {
        \__kernel_exp_not:w \exp_after:wN {#2}
        \__kernel_exp_not:w \exp_after:wN {#1}
      }
  }
\cs_new_protected:Npn \tl_gput_left:Nn #1#2
  {
    \__kernel_tl_gset:Nx #1
      { \__kernel_exp_not:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
  }
\cs_new_protected:Npn \tl_gput_left:NV #1#2
  {
    \__kernel_tl_gset:Nx #1
      { \exp_not:V #2 \__kernel_exp_not:w \exp_after:wN {#1} }
  }
\cs_new_protected:Npn \tl_gput_left:Nv #1#2
  {
    \__kernel_tl_gset:Nx #1
      { \exp_not:v {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
  }
\cs_new_protected:Npn \tl_gput_left:Ne #1#2
```

```
  {
    \__kernel_tl_gset:Nx #1
      {
        \__kernel_exp_not:w \tex_expanded:D { {#2} }
        \__kernel_exp_not:w \exp_after:wN {#1}
      }
  }
\cs_new_protected:Npn \tl_gput_left:No #1#2
  {
    \__kernel_tl_gset:Nx #1
      {
        \__kernel_exp_not:w \exp_after:wN {#2}
        \__kernel_exp_not:w \exp_after:wN {#1}
      }
  }
\cs_generate_variant:Nn \tl_put_left:Nn  { c }
\cs_generate_variant:Nn \tl_put_left:NV  { c }
\cs_generate_variant:Nn \tl_put_left:Nv  { c }
\cs_generate_variant:Nn \tl_put_left:Ne  { c }
\cs_generate_variant:Nn \tl_put_left:No  { c }
\cs_generate_variant:Nn \tl_put_left:Nn  { Nx, cx }
\cs_generate_variant:Nn \tl_gput_left:Nn { c }
\cs_generate_variant:Nn \tl_gput_left:NV { c }
\cs_generate_variant:Nn \tl_gput_left:Nv { c }
\cs_generate_variant:Nn \tl_gput_left:Ne { c }
\cs_generate_variant:Nn \tl_gput_left:No { c }
\cs_generate_variant:Nn \tl_gput_left:Nn { Nx , cx }
```

(*End of definition for* `\tl_put_left:Nn` *and* `\tl_gput_left:Nn`*. These functions are documented on page 112.*)

`\tl_put_right:Nn`
`\tl_put_right:NV`
`\tl_put_right:Nv`
`\tl_put_right:Ne`
`\tl_put_right:No`
`\tl_put_right:Nx`
`\tl_put_right:cn`
`\tl_put_right:cV`
`\tl_put_right:cv`
`\tl_put_right:ce`
`\tl_put_right:co`
`\tl_put_right:cx`
`\tl_gput_right:Nn`
`\tl_gput_right:NV`
`\tl_gput_right:Nv`
`\tl_gput_right:Ne`
`\tl_gput_right:No`
`\tl_gput_right:Nx`
`\tl_gput_right:cn`
`\tl_gput_right:cV`
`\tl_gput_right:cv`
`\tl_gput_right:ce`
`\tl_gput_right:co`
`\tl_gput_right:cx`

The same on the right.

```
\cs_new_protected:Npn \tl_put_right:Nn #1#2
  { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w \exp_after:wN { #1 #2 } } }
\cs_new_protected:Npn \tl_put_right:NV #1#2
  {
    \__kernel_tl_set:Nx #1
      { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:V #2 }
  }
\cs_new_protected:Npn \tl_put_right:Nv #1#2
  {
    \__kernel_tl_set:Nx #1
      { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:v {#2} }
  }
\cs_new_protected:Npn \tl_put_right:Ne #1#2
  {
    \__kernel_tl_set:Nx #1
      {
        \__kernel_exp_not:w \exp_after:wN {#1}
        \__kernel_exp_not:w \tex_expanded:D { {#2} }
      }
  }
\cs_new_protected:Npn \tl_put_right:No #1#2
  {
```

```
      \__kernel_tl_set:Nx #1
        {
          \__kernel_exp_not:w \exp_after:wN {#1}
          \__kernel_exp_not:w \exp_after:wN {#2}
        }
    }
\cs_new_protected:Npn \tl_gput_right:Nn #1#2
  { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w \exp_after:wN { #1 #2 } } }
\cs_new_protected:Npn \tl_gput_right:NV #1#2
  {
    \__kernel_tl_gset:Nx #1
      { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:V #2 }
  }
\cs_new_protected:Npn \tl_gput_right:Nv #1#2
  {
    \__kernel_tl_gset:Nx #1
      { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:v {#2} }
  }
\cs_new_protected:Npn \tl_gput_right:Ne #1#2
  {
    \__kernel_tl_gset:Nx #1
      {
        \__kernel_exp_not:w \exp_after:wN {#1}
        \__kernel_exp_not:w \tex_expanded:D { {#2} }
      }
  }
\cs_new_protected:Npn \tl_gput_right:No #1#2
  {
    \__kernel_tl_gset:Nx #1
      {
        \__kernel_exp_not:w \exp_after:wN {#1}
        \__kernel_exp_not:w \exp_after:wN {#2}
      }
  }
\cs_generate_variant:Nn \tl_put_right:Nn  { c }
\cs_generate_variant:Nn \tl_put_right:NV  { c }
\cs_generate_variant:Nn \tl_put_right:Nv  { c }
\cs_generate_variant:Nn \tl_put_right:Ne  { c }
\cs_generate_variant:Nn \tl_put_right:No  { c }
\cs_generate_variant:Nn \tl_put_right:Nn  { Nx , cx }
\cs_generate_variant:Nn \tl_gput_right:Nn { c }
\cs_generate_variant:Nn \tl_gput_right:NV { c }
\cs_generate_variant:Nn \tl_gput_right:Nv { c }
\cs_generate_variant:Nn \tl_gput_right:Ne { c }
\cs_generate_variant:Nn \tl_gput_right:No { c }
\cs_generate_variant:Nn \tl_gput_right:Nn { Nx, cx }
```

(*End of definition for* `\tl_put_right:Nn` *and* `\tl_gput_right:Nn`*. These functions are documented on page 113.*)

53.4 Internal quarks and quark-query functions

`\q__tl_nil`
`\q__tl_mark`
`\q__tl_stop`

Internal quarks.

```
\quark_new:N \q__tl_nil
\quark_new:N \q__tl_mark
\quark_new:N \q__tl_stop
```

(*End of definition for* `\q__tl_nil`*,* `\q__tl_mark`*, and* `\q__tl_stop`*.*)

`\q__tl_recursion_tail`
`\q__tl_recursion_stop`

Internal recursion quarks.

```
\quark_new:N \q__tl_recursion_tail
\quark_new:N \q__tl_recursion_stop
```

(*End of definition for* `\q__tl_recursion_tail` *and* `\q__tl_recursion_stop`*.*)

`\__tl_if_recursion_tail_break:nN`
`\__tl_if_recursion_tail_stop_p:n`
`\__tl_if_recursion_tail_stop:nTF`

Functions to query recursion quarks.

```
\__kernel_quark_new_test:N \__tl_if_recursion_tail_break:nN
\__kernel_quark_new_conditional:Nn \__tl_quark_if_nil:n { TF }
```

(*End of definition for* `\__tl_if_recursion_tail_break:nN` *and* `\__tl_if_recursion_tail_stop:nTF`*.*)

53.5 Reassigning token list category codes

`\c__tl_rescan_marker_tl`

The rescanning code needs a special token list containing the same character (chosen here to be a colon) with two different category codes: it cannot appear in the tokens being rescanned since all colons have the same category code.

```
\tl_const:Ne \c__tl_rescan_marker_tl { : \token_to_str:N : }
```

(*End of definition for* `\c__tl_rescan_marker_tl`*.*)

`\tl_set_rescan:Nnn`
`\tl_set_rescan:NnV`
`\tl_set_rescan:Nne`
`\tl_set_rescan:Nno`
`\tl_set_rescan:Nnx`
`\tl_set_rescan:cnn`
`\tl_set_rescan:cnV`
`\tl_set_rescan:cne`
`\tl_set_rescan:cno`
`\tl_set_rescan:cnx`
`\tl_gset_rescan:Nnn`
`\tl_gset_rescan:NnV`
`\tl_gset_rescan:Nne`
`\tl_gset_rescan:Nno`
`\tl_gset_rescan:Nnx`
`\tl_gset_rescan:cnn`
`\tl_gset_rescan:cnV`
`\tl_gset_rescan:cne`
`\tl_gset_rescan:cno`
`\tl_gset_rescan:cnx`
`\tl_rescan:nn`
`\tl_rescan:nV`
`\__tl_rescan_aux:`
`\__tl_set_rescan:NNnn`
`\__tl_set_rescan_multi:nNN`
`\__tl_rescan:NNw`

In a group, after some initial setup explained below and the user setup `#3` (followed by `\scan_stop:` to be safe), there is a call to `\__tl_set_rescan:nNN`. This shared auxiliary defined later distinguishes single-line and multi-line "files". In the simplest case of multi-line files, it calls (with the same arguments) `\__tl_set_rescan_multi:nNN`, whose code is included here to help understand the approach. This function rescans its argument `#1`, closes the group, and performs the assignment.

One difficulty when rescanning is that `\scantokens` treats the argument as a file, and without the correct settings a TeX error occurs:

```
! File ended while scanning definition of ...
```

A related minor issue is a warning due to opening a group before the `\scantokens` and closing it inside that temporary file; we avoid that by setting `\tracingnesting`. The standard solution to the "File ended" error is to grab the rescanned tokens as a delimited argument of an auxiliary, here `\__tl_rescan:NNw`, that performs the assignment, then let TeX "execute" the end of file marker. As usual in delimited arguments we use `\prg_do_nothing:` to avoid stripping an outer set braces: this is removed by using `o`-expanding assignments. The delimiter cannot appear within the rescanned token list because it contains twice the same character, with different catcodes.

For `\tl_rescan:nn` we cannot simply call `\__tl_set_rescan:NNnn \prg_do_nothing: \use:n` because that would leave the end-of-file marker *after* the result of rescanning. If that rescanned result is code that looks further in the input stream for arguments, it would break.

For multi-line files the only subtlety is that `\newlinechar` should be equal to `\endlinechar` because `\newlinechar` characters become new lines and then become

\endlinechar characters when writing to an abstract file and reading back. This equality is ensured by setting \newlinechar equal to \endlinechar. Prior to this, \endlinechar is set to -1 if it was 32 (in particular true after \ExplSyntaxOn) to avoid unreasonable line-breaks at every space for instance in error messages triggered by the user setup. Another side effect of reading back from the file is that spaces (catcode 10) are ignored at the beginning of lines, and spaces and tabs (character code 32 and 9) are ignored at the end of lines.

The two \if_false: ... \fi: are there to prevent alignment tabs to cause a change of tabular cell while rescanning. We put the "opening" one after \group_begin: so that if one accidentally f-expands \tl_set_rescan:Nnn braces remain balanced. This is essential in e-type arguments when \expanded is not available.

```
\cs_new_protected:Npn \tl_rescan:nn #1#2
  {
    \tl_set_rescan:Nnn \l__tl_internal_a_tl {#1} {#2}
    \exp_after:wN \__tl_rescan_aux:
    \l__tl_internal_a_tl
  }
\cs_generate_variant:Nn \tl_rescan:nn { nV }
\exp_args:NNo \cs_new_protected:Npn \__tl_rescan_aux:
  { \tl_clear:N \l__tl_internal_a_tl }
\cs_new_protected:Npn \tl_set_rescan:Nnn
  { \__tl_set_rescan:NNnn \tl_set:No }
\cs_new_protected:Npn \tl_gset_rescan:Nnn
  { \__tl_set_rescan:NNnn \tl_gset:No }
\cs_new_protected:Npn \__tl_set_rescan:NNnn #1#2#3#4
  {
    \group_begin:
      \if_false: { \fi:
      \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
      \int_compare:nNnT \tex_endlinechar:D = { 32 }
        { \int_set:Nn \tex_endlinechar:D { -1 } }
      \int_set_eq:NN \tex_newlinechar:D \tex_endlinechar:D
      #3 \scan_stop:
      \exp_args:No \__tl_set_rescan:nNN { \tl_to_str:n {#4} } #1 #2
    \if_false: } \fi:
  }
\cs_new_protected:Npn \__tl_set_rescan_multi:nNN #1#2#3
  {
    \tex_everyeof:D \exp_after:wN { \c__tl_rescan_marker_tl }
    \exp_after:wN \__tl_rescan:NNw
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN \prg_do_nothing:
    \tex_scantokens:D {#1}
  }
\exp_args:Nno \use:nn
  { \cs_new:Npn \__tl_rescan:NNw #1#2#3 } \c__tl_rescan_marker_tl
  {
    \group_end:
    #1 #2 {#3}
  }
\cs_generate_variant:Nn \tl_set_rescan:Nnn  { NnV , Nne , c , cnV , cne }
\cs_generate_variant:Nn \tl_set_rescan:Nnn  { Nno , Nnx , cno , cnx }
```

```
\cs_generate_variant:Nn \tl_gset_rescan:Nnn { NnV , Nne , c , cnV , cne }
\cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno , Nnx , cno , cnx }
```

(*End of definition for* `\tl_set_rescan:Nnn` *and others. These functions are documented on page 126.*)

`\__tl_set_rescan:nNN`
`\__tl_set_rescan_single:nnNN`
`\__tl_set_rescan_single_aux:nnnNN`
`\__tl_set_rescan_single_aux:w`

The function `\__tl_set_rescan:nNN` calls `\__tl_set_rescan_multi:nNN` or `\__tl_set_rescan_single:nnNN { ' }` depending on whether its argument is a single-line fragment of code/data or is made of multiple lines by testing for the presence of a `\newlinechar` character. If `\newlinechar` is out of range, the argument is assumed to be a single line.

For a single line, no `\endlinechar` should be added, so it is set to -1, and spaces should not be removed. Trailing spaces and tabs are a difficult matter, as TeX removes these at a very low level. The only way to preserve them is to rescan not the argument but the argument followed by a character with a reasonable category code. Here, 11 (letter) and 12 (other) are accepted, as these are convenient, suitable for delimiting an argument, and it is very unlikely that none of the ASCII characters are in one of these categories. To avoid selecting one particular character to put at the end, whose category code may have been modified, there is a loop through characters from `'` (ASCII 39) to `~` (ASCII 127). The choice of starting point was made because this is the start of a very long range of characters whose standard category is letter or other, thus minimizing the number of steps needed by the loop (most often just a single one). If no valid character is found (very rare), fall-back on `\__tl_set_rescan_multi:nNN`.

Otherwise, once a valid character is found (let us use `'` in this explanation) run some code very similar to `\__tl_set_rescan_multi:nNN` but with `'` added at both ends of the input. Of course, we need to define the auxiliary `\__tl_set_rescan_single:NNww` on the fly to remove the additional `'` that is just before `::` (by which we mean `\c__tl_rescan_marker_tl`). Note that the argument must be delimited by `'` with the current catcode; this is done thanks to `\char_generate:nn`. Yet another issue is that the rescanned token list may contain a comment character, in which case the `'` we expected is not there. We fix this as follows: rather than just `::` we set `\everyeof` to `::{`⟨*code1*⟩`} '::{`⟨*code2*⟩`} \s__tl_stop`. The auxiliary `\__tl_set_rescan_single:NNww` runs the `o`-expanding assignment, expanding either ⟨*code1*⟩ or ⟨*code2*⟩ before its the main argument `#3`. In the typical case without comment character, ⟨*code1*⟩ is expanded, removing the leading `'`. In the rarer case with comment character, ⟨*code2*⟩ is expanded, calling `\__tl_set_rescan_single_aux:w`, which removes the trailing `::{`⟨*code1*⟩`}` and the leading `'`.

```
\cs_new_protected:Npn \__tl_set_rescan:nNN #1
  {
    \int_compare:nNnTF \tex_newlinechar:D < 0
      { \use_ii:nn }
      {
        \exp_args:Nnf \tl_if_in:nnTF {#1}
          { \char_generate:nn { \tex_newlinechar:D } { 12 } }
      }
        { \__tl_set_rescan_multi:nNN }
        {
          \int_set:Nn \tex_endlinechar:D { -1 }
          \__tl_set_rescan_single:nnNN { `' }
        }
    {#1}
  }
\cs_new_protected:Npn \__tl_set_rescan_single:nnNN #1
```

```
  {
    \int_compare:nNnTF
      { \char_value_catcode:n {#1} / 2 } = 6
      {
        \exp_args:Nof \__tl_set_rescan_single_aux:nnnNN
          \c__tl_rescan_marker_tl
          { \char_generate:nn {#1} { \char_value_catcode:n {#1} } }
      }
      {
        \int_compare:nNnTF {#1} < { `\~ }
          {
            \exp_args:Nf \__tl_set_rescan_single:nnNN
              { \int_eval:n { #1 + 1 } }
          }
          { \__tl_set_rescan_multi:nNN }
      }
  }
\cs_new_protected:Npn \__tl_set_rescan_single_aux:nnnNN #1#2#3#4#5
  {
    \tex_everyeof:D
      {
        #1 \use_none:n
        #2 #1 { \exp:w \__tl_set_rescan_single_aux:w }
        \s__tl_stop
      }
    \cs_set:Npn \__tl_rescan:NNw ##1##2##3 #2 #1 ##4 ##5 \s__tl_stop
      {
        \group_end:
        ##1 ##2 { ##4 ##3 }
      }
    \exp_after:wN \__tl_rescan:NNw
    \exp_after:wN #4
    \exp_after:wN #5
    \tex_scantokens:D { #2 #3 #2 }
  }
\exp_args:Nno \use:nn
  { \cs_new:Npn \__tl_set_rescan_single_aux:w #1 }
  \c__tl_rescan_marker_tl #2
  { \use_i:nn \exp_end: #1 }
```

(*End of definition for* `\__tl_set_rescan:nNN` *and others.*)

53.6 Modifying token list variables

`\tl_replace_once:Nnn`
`\tl_replace_once:NVn`
`\tl_replace_once:NnV`
`\tl_replace_once:Nen`
`\tl_replace_once:Nne`
`\tl_replace_once:Nee`
`\tl_replace_once:Nxn`
`\tl_replace_once:Nnx`
`\tl_replace_once:Nxx`
`\tl_replace_once:cnn`
`\tl_replace_once:cVn`
`\tl_replace_once:cnV`
`\tl_replace_once:cen`
`\tl_replace_once:cne`
`\tl_replace_once:cee`
`\tl_replace_once:cxn`
`\tl_replace_once:cnx`
`\tl_replace_once:cxx`
`\tl_greplace_once:Nnn`

All of the `replace` functions call `\__tl_replace:NnNNNnn` with appropriate arguments. The first two arguments are explained later. The next controls whether the replacement function calls itself (`\__tl_replace_next:w`) or stops (`\__tl_replace_wrap:w`) after the first replacement. Next comes an `e`-type assignment function `\tl_set:Ne` or `\tl_gset:Ne` for local or global replacements. Finally, the three arguments ⟨*tl var*⟩ {⟨*pattern*⟩} {⟨*replacement*⟩} provided by the user. When describing the auxiliary functions below, we denote the contents of the ⟨*tl var*⟩ by ⟨*token list*⟩.

```
\cs_new_protected:Npn \tl_replace_once:Nnn
  { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_wrap:w \__kernel_tl_set:Nx }
```

```
\cs_new_protected:Npn \tl_greplace_once:Nnn
  { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_wrap:w \__kernel_tl_gset:Nx }
\cs_new_protected:Npn \tl_replace_all:Nnn
  { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_next:w \__kernel_tl_set:Nx  }
\cs_new_protected:Npn \tl_greplace_all:Nnn
  { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_next:w \__kernel_tl_gset:Nx }
\cs_generate_variant:Nn \tl_replace_once:Nnn
  { NnV , Nne , NV , Ne , Nee , c , cnV , cne , cV , ce , cee }
\cs_generate_variant:Nn \tl_replace_once:Nnn
  { Nx , Nnx , Nxx , cxn , cnx , cxx }
\cs_generate_variant:Nn \tl_greplace_once:Nnn
  { NnV , Nne , NV , Ne , Nee , c , cnV , cne , cV , ce , cee }
\cs_generate_variant:Nn \tl_greplace_once:Nnn
  { Nx , Nnx , Nxx , cxn , cnx , cxx }
\cs_generate_variant:Nn \tl_replace_all:Nnn
  { NnV , Nne , NV , Ne , Nee , c , cnV , cne , cV , ce , cee }
\cs_generate_variant:Nn \tl_replace_all:Nnn
  { Nx , Nnx , Nxx , cxn , cnx , cxx }
\cs_generate_variant:Nn \tl_greplace_all:Nnn
  { NnV , Nne , NV , Ne , Nee , c , cnV , cne , cV , ce , cee }
\cs_generate_variant:Nn \tl_greplace_all:Nnn
  { Nx , Nnx , Nxx , cxn , cnx , cxx }
```

(*End of definition for* `\tl_replace_once:Nnn` *and others. These functions are documented on page* *124*.)

`\__tl_replace:NnNNNnn`
`\__tl_replace_auxi:NnnNNNnn`
`\__tl_replace_auxii:nNNNnn`
`\__tl_replace_next:w`
`\__tl_replace_next_aux:w`
`\__tl_replace_wrap:w`

To implement the actual replacement auxiliary `\__tl_replace_auxii:nNNNnn` we need a ⟨*delimiter*⟩ with the following properties:

- all occurrences of the ⟨*pattern*⟩ `#6` in "⟨*token list*⟩ ⟨*delimiter*⟩" belong to the ⟨*token list*⟩ and have no overlap with the ⟨*delimiter*⟩,
- the first occurrence of the ⟨*delimiter*⟩ in "⟨*token list*⟩ ⟨*delimiter*⟩" is the trailing ⟨*delimiter*⟩.

We first find the building blocks for the ⟨*delimiter*⟩, namely two tokens ⟨*A*⟩ and ⟨*B*⟩ such that ⟨*A*⟩ does not appear in `#6` and `#6` is not ⟨*B*⟩ (this condition is trivial if `#6` has more than one token). Then we consider the delimiters "⟨*A*⟩" and "⟨*A*⟩ ⟨*A*⟩^{n} ⟨*B*⟩ ⟨*A*⟩^{n} ⟨*B*⟩", for $n \geq 1$, where ⟨*A*⟩^{n} denotes n copies of ⟨*A*⟩, and we choose as our ⟨*delimiter*⟩ the first one which is not in the ⟨*token list*⟩.

Every delimiter in the set obeys the first condition: `#6` does not contain ⟨*A*⟩ hence cannot be overlapping with the ⟨*token list*⟩ and the ⟨*delimiter*⟩, and it cannot be within the ⟨*delimiter*⟩ since it would have to be in one of the two ⟨*B*⟩ hence be equal to this single token (or empty, but this is an error case filtered separately). Given the particular form of these delimiters, for which no prefix is also a suffix, the second condition is actually a consequence of the weaker condition that the ⟨*delimiter*⟩ we choose does not appear in the ⟨*token list*⟩. Additionally, the set of delimiters is such that a ⟨*token list*⟩ of n tokens can contain at most $O(n^{1/2})$ of them, hence we find a ⟨*delimiter*⟩ with at most $O(n^{1/2})$ tokens in a time at most $O(n^{3/2})$. Bear in mind that these upper bounds are reached only in very contrived scenarios: we include the case "⟨*A*⟩" in the list of delimiters to try, so that the ⟨*delimiter*⟩ is simply `\q__tl_mark` in the most common situation where neither the ⟨*token list*⟩ nor the ⟨*pattern*⟩ contains `\q__tl_mark`.

Let us now ahead, optimizing for this most common case. First, two special cases: an empty ⟨*pattern*⟩ `#6` is an error, and if `#1` is absent from both the ⟨*token list*⟩ `#5` and the ⟨*pattern*⟩ `#6` then we can use it as the ⟨*delimiter*⟩ through `\__tl_replace_auxii:nNNNnn {#1}`. Otherwise, we end up calling `\__tl_replace:NnNNNnn` repeatedly with the first two arguments `\q__tl_mark {?}`, `\? {??}`, `\?? {???}`, and so on, until `#6` does not contain the control sequence `#1`, which we take as our ⟨*A*⟩. The argument `#2` only serves to collect `?` characters for `#1`. Note that the order of the tests means that the first two are done every time, which is wasteful (for instance, we repeatedly test for the emptyness of `#6`). However, this is rare enough not to matter. Finally, choose ⟨*B*⟩ to be `\q__tl_nil` or `\q__tl_stop` such that it is not equal to `#6`.

The `\__tl_replace_auxi:NnnNNNnn` auxiliary receives `{`$\langle A\rangle$`}` and `{`$\langle A\rangle^n\langle B\rangle$`}` as its arguments, initially with $n = 1$. If "$\langle A\rangle\ \langle A\rangle^n\langle B\rangle\ \langle A\rangle^n\langle B\rangle$" is in the ⟨*token list*⟩ then increase n and try again. Once it is not anymore in the ⟨*token list*⟩ we take it as our ⟨*delimiter*⟩ and pass this to the `auxii` auxiliary.

```
\cs_new_protected:Npn \__tl_replace:NnNNNnn #1#2#3#4#5#6#7
  {
    \tl_if_empty:nTF {#6}
      {
        \msg_error:nne { kernel } { empty-search-pattern }
          { \tl_to_str:n {#7} }
      }
      {
        \tl_if_in:onTF { #5 #6 } {#1}
          {
            \tl_if_in:nnTF {#6} {#1}
              { \exp_args:Nc \__tl_replace:NnNNNnn {#2} {#2?} }
              {
                \__tl_quark_if_nil:nTF {#6}
                  { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q__tl_stop } }
                  { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q__tl_nil  } }
              }
          }
          { \__tl_replace_auxii:nNNNnn {#1} }
          #3#4#5 {#6} {#7}
      }
  }
\cs_new_protected:Npn \__tl_replace_auxi:NnnNNNnn #1#2#3
  {
    \tl_if_in:NnTF #1 { #2 #3 #3 }
      { \__tl_replace_auxi:NnnNNNnn #1 { #2 #3 } {#2} }
      { \__tl_replace_auxii:nNNNnn { #2 #3 #3 } }
  }
```

The auxiliary `\__tl_replace_auxii:nNNNnn` receives the following arguments:

> `{`⟨*delimiter*⟩`}` ⟨*function*⟩ ⟨*assignment*⟩
> ⟨*tl var*⟩ `{`⟨*pattern*⟩`}` `{`⟨*replacement*⟩`}`

All of its work is done between `\group_align_safe_begin:` and `\group_align_safe_end:` to avoid issues in alignments. It does the actual replacement within `#3 #4 {...}`, an e-expanding ⟨*assignment*⟩ `#3` to the ⟨*tl var*⟩ `#4`. The auxiliary `\__tl_replace_next:w` is called, followed by the ⟨*token list*⟩, some tokens including the ⟨*delimiter*⟩ `#1`, followed by the ⟨*pattern*⟩ `#5`. This auxiliary finds an argument delimited by `#5` (the presence of a trailing `#5` avoids runaway arguments) and calls `\__tl_replace_wrap:w` to test whether this `#5` is found within the ⟨*token list*⟩ or is the trailing one.

If on the one hand it is found within the ⟨*token list*⟩, then ##1 cannot contain the ⟨*delimiter*⟩ #1 that we worked so hard to obtain, thus __tl_replace_wrap:w gets ##1 as its own argument ##1, and protects it against the e-expanding assignment. It also finds \exp_not:n as ##2 and does nothing to it, thus letting through \exp_not:n {⟨*replacement*⟩} into the assignment. Note that __tl_replace_next:w and __tl_-replace_wrap:w are always called followed by two empty brace groups. These are safe because no delimiter can match them. They prevent losing braces when grabbing delimited arguments, but require the use of \exp_not:o and \use_none:nn, rather than simply \exp_not:n. Afterwards, __tl_replace_next:w is called to repeat the replacement, or __tl_replace_wrap:w if we only want a single replacement. In this second case, ##1 is the ⟨*remaining tokens*⟩ in the ⟨*token list*⟩ and ##2 is some ⟨*ending code*⟩ which ends the assignment and removes the trailing tokens #5 using some \if_false: { \fi: } trickery because #5 may contain any delimiter.

If on the other hand the argument ##1 of __tl_replace_next:w is delimited by the trailing ⟨*pattern*⟩ #5, then ##1 is "{ } { } ⟨*token list*⟩ ⟨*delimiter*⟩ {⟨*ending code*⟩}", hence __tl_replace_wrap:w finds "{ } { } ⟨*token list*⟩" as ##1 and the ⟨*ending code*⟩ as ##2. It leaves the ⟨*token list*⟩ into the assignment and unbraces the ⟨*ending code*⟩ which removes what remains (essentially the ⟨*delimiter*⟩ and ⟨*replacement*⟩).

```
\cs_new_protected:Npn \__tl_replace_auxii:nNNNnn #1#2#3#4#5#6
  {
    \group_align_safe_begin:
    \cs_set:Npn \__tl_replace_wrap:w ##1 #1 ##2
      { \__kernel_exp_not:w \exp_after:wN { \use_none:nn ##1 } ##2 }
    \cs_set:Npe \__tl_replace_next:w ##1 #5
      {
        \exp_not:N \__tl_replace_wrap:w ##1
        \exp_not:n { #1 }
        \exp_not:n { \exp_not:n {#6} }
        \exp_not:n { #2 { } { } }
      }
    #3 #4
      {
        \exp_after:wN \__tl_replace_next_aux:w
        #4
        #1
        {
          \if_false: { \fi: }
          \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
        }
        #5
      }
    \group_align_safe_end:
  }
\cs_new:Npn \__tl_replace_next_aux:w { \__tl_replace_next:w { } { } }
\cs_new_eq:NN \__tl_replace_wrap:w ?
\cs_new_eq:NN \__tl_replace_next:w ?
```

(*End of definition for* __tl_replace:NnNNNnn *and others.*)

\tl_remove_once:Nn
\tl_remove_once:NV
\tl_remove_once:Ne
\tl_remove_once:cn
\tl_remove_once:cV
\tl_remove_once:ce
\tl_gremove_once:Nn
\tl_gremove_once:NV
\tl_gremove_once:cn
\tl_gremove_once:cV

Removal is just a special case of replacement.

```
\cs_new_protected:Npn \tl_remove_once:Nn #1#2
  { \tl_replace_once:Nnn #1 {#2} { } }
\cs_new_protected:Npn \tl_gremove_once:Nn #1#2
```

```
  { \tl_greplace_once:Nnn #1 {#2} { } }
\cs_generate_variant:Nn \tl_remove_once:Nn  { NV , Ne , c , cV , ce }
\cs_generate_variant:Nn \tl_gremove_once:Nn { NV , Ne , c , cV , ce }
```

(*End of definition for* `\tl_remove_once:Nn` *and* `\tl_gremove_once:Nn`*. These functions are documented on page 125.*)

`\tl_remove_all:Nn`
`\tl_remove_all:NV`
`\tl_remove_all:Ne`
`\tl_remove_all:Nx`
`\tl_remove_all:cn`
`\tl_remove_all:cV`
`\tl_remove_all:ce`
`\tl_remove_all:cx`
`\tl_gremove_all:Nn`
`\tl_gremove_all:NV`
`\tl_gremove_all:Ne`
`\tl_gremove_all:Nx`
`\tl_gremove_all:cn`
`\tl_gremove_all:cV`
`\tl_gremove_all:ce`
`\tl_gremove_all:cx`

Removal is just a special case of replacement.

```
\cs_new_protected:Npn \tl_remove_all:Nn #1#2
  { \tl_replace_all:Nnn #1 {#2} { } }
\cs_new_protected:Npn \tl_gremove_all:Nn #1#2
  { \tl_greplace_all:Nnn #1 {#2} { } }
\cs_generate_variant:Nn \tl_remove_all:Nn  { NV , Ne , c , cV , ce }
\cs_generate_variant:Nn \tl_remove_all:Nn  { Nx , cx }
\cs_generate_variant:Nn \tl_gremove_all:Nn { NV , Ne , c , cV , ce }
\cs_generate_variant:Nn \tl_gremove_all:Nn { Nx , cx }
```

(*End of definition for* `\tl_remove_all:Nn` *and* `\tl_gremove_all:Nn`*. These functions are documented on page 125.*)

53.7 Token list conditionals

`\tl_if_empty_p:N`
`\tl_if_empty_p:c`
`\tl_if_empty:NTF`
`\tl_if_empty:cTF`

These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).

```
\prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \c_empty_tl
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_generate_conditional_variant:Nnn \tl_if_empty:N
  { c } { p , T , F , TF }
```

(*End of definition for* `\tl_if_empty:NTF`*. This function is documented on page 113.*)

`\tl_if_empty_p:n`
`\tl_if_empty_p:V`
`\tl_if_empty_p:e`
`\tl_if_empty:nTF`
`\tl_if_empty:VTF`
`\tl_if_empty:eTF`

The `\if:w` triggers the expansion of `\tl_to_str:n` which converts the argument to a string: this is empty if and only if the argument is. Then `\if:w \scan_stop: ... \scan_stop:` is `true` if and only if the string `...` is empty. It could be tempting to use `\if:w \scan_stop: #1 \scan_stop:` directly. But this fails on a token list expanding to anything starting with `\scan_stop:` leaving everything that follows in the input stream.

```
\prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
  {
    \if:w \scan_stop: \tl_to_str:n {#1} \scan_stop:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_generate_conditional_variant:Nnn \tl_if_empty:n
  { V , e } { p , TF , T , F }
```

(*End of definition for* `\tl_if_empty:nTF`. *This function is documented on page 113.*)

`\tl_if_empty_p:o`
`\tl_if_empty:oTF`
`\__tl_if_empty_if:o`

The auxiliary function `\__tl_if_empty_if:o` is for use in various token list conditionals which reduce to testing if a given token list is empty after applying a simple function to it. The test for emptiness is based on `\tl_if_empty:nTF`, but the expansion is hard-coded for efficiency, as this auxiliary function is used in several places. We don't put `\prg_return_true:` and so on in the definition of the auxiliary, because that would prevent an optimization applied to conditionals that end with this code. Also the `\@@_if_empty_if:o` is expanded once in `\tl_if_empty:oTF` for efficiency as well (and to reduce code doubling).

```
\cs_new:Npn \__tl_if_empty_if:o #1
  {
    \if:w \scan_stop: \__kernel_tl_to_str:w \exp_after:wN {#1} \scan_stop:
  }
\exp_args:Nno \use:n
  { \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F } }
  {
    \__tl_if_empty_if:o {#1}
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\tl_if_empty:nTF` *and* `\__tl_if_empty_if:o`. *This function is documented on page 113.*)

`\tl_if_blank_p:n`
`\tl_if_blank_p:V`
`\tl_if_blank_p:o`
`\tl_if_blank:nTF`
`\tl_if_blank:VTF`
`\tl_if_blank:oTF`
`\__tl_if_blank_p:NNw`

TeX skips spaces when reading a non-delimited arguments. Thus, a ⟨*token list*⟩ is blank if and only if `\use_none:n` ⟨*token list*⟩ `?` is empty after one expansion. The auxiliary `\__tl_if_empty_if:o` is a fast emptyness test, converting its argument to a string (after one expansion) and using the test `\if:w \scan_stop: ... \scan_stop:`.

```
\exp_args:Nno \use:n
  { \prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF } }
  {
    \__tl_if_empty_if:o { \use_none:n #1 ? }
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_generate_conditional_variant:Nnn \tl_if_blank:n
  { e , V , o } { p , T , F , TF }
```

(*End of definition for* `\tl_if_blank:nTF` *and* `\__tl_if_blank_p:NNw`. *This function is documented on page 113.*)

`\tl_if_eq_p:NN`
`\tl_if_eq_p:Nc`
`\tl_if_eq_p:cN`
`\tl_if_eq_p:cc`
`\tl_if_eq:NNTF`
`\tl_if_eq:NcTF`
`\tl_if_eq:cNTF`
`\tl_if_eq:ccTF`

Returns `\c_true_bool` if and only if the two token list variables are equal.

```
\prg_new_eq_conditional:NNn \tl_if_eq:NN \cs_if_eq:NN { p , T , F , TF }
\prg_generate_conditional_variant:Nnn \tl_if_eq:NN
  { Nc , c , cc } { p , TF , T , F }
```

(*End of definition for* `\tl_if_eq:NNTF`. *This function is documented on page 113.*)

\l__tl_internal_a_tl
\l__tl_internal_b_tl

Temporary storage.

```
\tl_new:N \l__tl_internal_a_tl
\tl_new:N \l__tl_internal_b_tl
```

(*End of definition for* \l__tl_internal_a_tl *and* \l__tl_internal_b_tl.)

\tl_if_eq:Nn*TF*

A simple store and compare routine.

```
\prg_new_protected_conditional:Npnn \tl_if_eq:Nn #1#2 { T , F , TF }
  {
    \group_begin:
      \tl_set:Nn \l__tl_internal_b_tl {#2}
      \exp_after:wN
    \group_end:
    \if_meaning:w #1 \l__tl_internal_b_tl
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_generate_conditional_variant:Nnn \tl_if_eq:Nn { c } { TF , T , F }
```

(*End of definition for* \tl_if_eq:NnTF. *This function is documented on page* *113*.)

\tl_if_eq:nn*TF*
\tl_if_eq:nV*TF*
\tl_if_eq:ne*TF*
\tl_if_eq:Vn*TF*
\tl_if_eq:en*TF*
\tl_if_eq:ee*TF*
\tl_if_eq:xn*TF*
\tl_if_eq:nx*TF*
\tl_if_eq:xx*TF*

A simple store and compare routine.

```
\prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F ,  TF }
  {
    \group_begin:
      \tl_set:Nn \l__tl_internal_a_tl {#1}
      \tl_set:Nn \l__tl_internal_b_tl {#2}
      \exp_after:wN
    \group_end:
    \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_generate_conditional_variant:Nnn \tl_if_eq:nn
  { nV , ne , nx , V, e , ee , x , xx }
  { TF , T , F }
```

(*End of definition for* \tl_if_eq:nnTF. *This function is documented on page* *114*.)

\tl_if_in:Nn*TF*
\tl_if_in:NV*TF*
\tl_if_in:No*TF*
\tl_if_in:cn*TF*
\tl_if_in:cV*TF*
\tl_if_in:co*TF*

See \tl_if_in:nnTF for further comments. Here we simply expand the token list variable and pass it to \tl_if_in:nnTF.

```
\cs_new_protected:Npn \tl_if_in:NnT  { \exp_args:No \tl_if_in:nnT  }
\cs_new_protected:Npn \tl_if_in:NnF  { \exp_args:No \tl_if_in:nnF  }
\cs_new_protected:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
\prg_generate_conditional_variant:Nnn \tl_if_in:Nn
  { NV , No , c , cV , co } { T , F , TF }
```

(*End of definition for* \tl_if_in:NnTF. *This function is documented on page* *114*.)

\tl_if_in:nnTF
\tl_if_in:VnTF
\tl_if_in:VVTF
\tl_if_in:onTF
\tl_if_in:ooTF
\tl_if_in:nVTF
\tl_if_in:noTF

Once more, the test relies on the emptiness test for robustness. The function `\__tl_tmp:w` removes tokens until the first occurrence of `#2`. If this does not appear in `#1`, then the final `#2` is removed, leaving an empty token list. Otherwise some tokens remain, and the test is `false`. See `\tl_if_empty:nTF` for details on the emptiness test.

Treating correctly cases like `\tl_if_in:nnTF {a state}{states}`, where `#1#2` contains `#2` before the end, requires special care. To cater for this case, we insert `{}{}` between the two token lists. This marker may not appear in `#2` because of TEX limitations on what can delimit a parameter, hence we are safe. Using two brace groups makes the test work also for empty arguments. The `\if_false:` constructions are a faster way to do `\group_align_safe_begin:` and `\group_align_safe_end:`. The `\scan_stop:` ensures that f-expanding `\tl_if_in:nnTF` does not lead to unbalanced braces.

```
\prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T  , F , TF }
  {
    \scan_stop:
    \if_false: { \fi:
    \cs_set:Npn \__tl_tmp:w ##1 #2 { }
    \tl_if_empty:oTF { \__tl_tmp:w #1 {} {} #2 }
      { \prg_return_false: } { \prg_return_true: }
    \if_false: } \fi:
  }
\prg_generate_conditional_variant:Nnn \tl_if_in:nn
  { V , VV , o , oo , nV , no } { T , F , TF }
```

(*End of definition for* `\tl_if_in:nnTF`. *This function is documented on page 114.*)

\tl_if_novalue_p:n
\tl_if_novalue:nTF
__tl_if_novalue:w

Tests whether `##1` matches `-NoValue-` exactly (with suitable catcodes): this is similar to `\quark_if_nil:nTF`. The first argument of `\__tl_if_novalue:w` is empty if and only if `##1` starts with `-NoValue-`, while the second argument is empty if `##1` is exactly `-NoValue-` or if it has a question mark just following `-NoValue-`. In this second case, however, the material after the first `?!` remains and makes the emptyness test return `false`.

```
\cs_set_protected:Npn \__tl_tmp:w #1
  {
    \prg_new_conditional:Npnn \tl_if_novalue:n ##1
      { p , T ,  F , TF }
      {
        \__tl_if_empty_if:o { \__tl_if_novalue:w {} ##1 {} ? ! #1 ? ? ! }
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      }
    \cs_new:Npn \__tl_if_novalue:w ##1 #1 ##2 ? ##3 ? ! { ##1 ##2 }
  }
\exp_args:No \__tl_tmp:w { \c_novalue_tl }
```

(*End of definition for* `\tl_if_novalue:nTF` *and* `\__tl_if_novalue:w`. *This function is documented on page 114.*)

\tl_if_single_p:N
\tl_if_single_p:c
\tl_if_single:NTF
\tl_if_single:cTF

Expand the token list and feed it to `\tl_if_single:nTF`.

```
\cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
\cs_new:Npn \tl_if_single:NT  { \exp_args:No \tl_if_single:nT  }
\cs_new:Npn \tl_if_single:NF  { \exp_args:No \tl_if_single:nF  }
```

```
\cs_new:Npn \tl_if_single:NTF { \exp_args:No \tl_if_single:nTF }
\prg_generate_conditional_variant:Nnn \tl_if_single:N {c} { p , T , F , TF }
```

(*End of definition for* `\tl_if_single:NTF`. *This function is documented on page 114.*)

`\tl_if_single_p:n`
`\tl_if_single:nTF`
`\__tl_if_single:nnw`

This test is similar to `\tl_if_empty:nTF`. Expanding `\use_none:nn #1 ??` once yields an empty result if `#1` is blank, a single `?` if `#1` has a single item, and otherwise yields some tokens ending with `??`. Then, `\__kernel_tl_to_str:w` makes sure there are no odd category codes. An earlier version would compare the result to a single `?` using string comparison, but the Lua call is slow in LuaTEX. Instead, `\__tl_if_single:nnw` picks the second token in front of it. If `#1` is empty, this token is the trailing `?` and the `\if:w` test yields `false`. If `#1` has a single item, the token is `\scan_stop:` and the `\if:w` test yields `true`. Otherwise, it is one of the characters resulting from `\tl_to_str:n`, and the `\if:w` test yields `false`. Note that `\if:w` and `\__kernel_tl_to_str:w` are primitives that take care of expansion.

```
\prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
  {
    \if:w \scan_stop: \exp_after:wN \__tl_if_single:nnw
        \__kernel_tl_to_str:w
          \exp_after:wN { \use_none:nn #1 ?? } \scan_stop: ? \s__tl_stop
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \__tl_if_single:nnw #1#2#3 \s__tl_stop {#2}
```

(*End of definition for* `\tl_if_single:nTF` *and* `\__tl_if_single:nnw`. *This function is documented on page 114.*)

`\tl_if_single_token_p:n`
`\tl_if_single_token:nTF`

There are four cases: empty token list, token list starting with a normal token, with a brace group, or with a space token. If the token list starts with a normal token, remove it and check for emptiness. For the next case, an empty token list is not a single token. Finally, we have a non-empty token list starting with a space or a brace group. Applying `f`-expansion yields an empty result if and only if the token list is a single space.

```
\prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
  {
    \tl_if_head_is_N_type:nTF {#1}
      { \__tl_if_empty_if:o { \use_none:n #1 } }
      {
        \tl_if_empty:nTF {#1}
          { \if_false: }
          { \__tl_if_empty_if:o { \exp:w \exp_end_continue_f:w #1 } }
      }
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\tl_if_single_token:nTF`. *This function is documented on page 114.*)

53.8 Mapping over token lists

\tl_map_function:nN
\tl_map_function:NN
\tl_map_function:cN
__tl_map_function:Nnnnnnnnn
__tl_map_function_end:w
__tl_use_none_delimit_by_s_stop:w

Expandable loop macro for token lists. We use the internal scan mark `\s__tl_stop` (defined later), which is not allowed to show up in the token list `#1` since it is internal to l3tl. This allows us a very fast test of whether some ⟨*item*⟩ is the end-marker `\s__tl_stop`, namely call `\__tl_use_none_delimit_by_s_stop:w` ⟨*item*⟩ ⟨*function*⟩ `\s__tl_stop`, which calls ⟨*function*⟩ if the ⟨*item*⟩ is the end-marker. To speed up the loop even more, only test one out of eight items, and once we hit one of the eight end-markers, go more slowly through the last few items of the list using `\__tl_map_function_end:w`.

```
\cs_new:Npn \tl_map_function:nN #1#2
  {
    \__tl_map_function:Nnnnnnnnn #2 #1
      \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
      \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
    \prg_break_point:Nn \tl_map_break: { }
  }
\cs_new:Npn \tl_map_function:NN
  { \exp_args:No \tl_map_function:nN }
\cs_generate_variant:Nn \tl_map_function:NN { c }
\cs_new:Npn \__tl_map_function:Nnnnnnnnn #1#2#3#4#5#6#7#8#9
  {
    \__tl_use_none_delimit_by_s_stop:w
      #9 \__tl_map_function_end:w \s__tl_stop
    #1 {#2} #1 {#3} #1 {#4} #1 {#5} #1 {#6} #1 {#7} #1 {#8} #1 {#9}
    \__tl_map_function:Nnnnnnnnn #1
  }
\cs_new:Npn \__tl_map_function_end:w \s__tl_stop #1#2
  {
    \__tl_use_none_delimit_by_s_stop:w #2 \tl_map_break: \s__tl_stop
    #1 {#2}
    \__tl_map_function_end:w \s__tl_stop
  }
\cs_new:Npn \__tl_use_none_delimit_by_s_stop:w #1 \s__tl_stop { }
```

(*End of definition for* `\tl_map_function:nN` *and others. These functions are documented on page* *119*.)

\tl_map_inline:nn
\tl_map_inline:Nn
\tl_map_inline:cn

The inline functions are straight forward by now. We use a little trick with the counter `\g__kernel_prg_map_int` to make them nestable. We can also make use of `\__tl_map_function:Nnnnnnnnn` from before.

```
\cs_new_protected:Npn \tl_map_inline:nn #1#2
  {
    \int_gincr:N \g__kernel_prg_map_int
    \cs_gset_protected:cpn
      { __tl_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
    \exp_args:Nc \__tl_map_function:Nnnnnnnnn
      { __tl_map_ \int_use:N \g__kernel_prg_map_int :w }
      #1
      \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
      \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
    \prg_break_point:Nn \tl_map_break:
      { \int_gdecr:N \g__kernel_prg_map_int }
  }
\cs_new_protected:Npn \tl_map_inline:Nn
```

```
  { \exp_args:No \tl_map_inline:nn }
\cs_generate_variant:Nn \tl_map_inline:Nn { c }
```

(*End of definition for* `\tl_map_inline:nn` *and* `\tl_map_inline:Nn`. *These functions are documented on page 119.*)

`\tl_map_tokens:nn`
`\tl_map_tokens:Nn`
`\tl_map_tokens:cn`
`\__tl_map_tokens:nnnnnnnnn`
`\__tl_map_tokens_end:w`

Much like the function mapping.

```
\cs_new:Npn \tl_map_tokens:nn #1#2
  {
    \__tl_map_tokens:nnnnnnnnn {#2} #1
      \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
      \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
    \prg_break_point:Nn \tl_map_break: { }
  }
\cs_new:Npn \tl_map_tokens:Nn
  { \exp_args:No \tl_map_tokens:nn }
\cs_generate_variant:Nn \tl_map_tokens:Nn { c }
\cs_new:Npn \__tl_map_tokens:nnnnnnnnn #1#2#3#4#5#6#7#8#9
  {
    \__tl_use_none_delimit_by_s_stop:w
      #9 \__tl_map_tokens_end:w \s__tl_stop
    \use:n {#1} {#2} \use:n {#1} {#3} \use:n {#1} {#4} \use:n {#1} {#5}
    \use:n {#1} {#6} \use:n {#1} {#7} \use:n {#1} {#8} \use:n {#1} {#9}
    \__tl_map_tokens:nnnnnnnnn {#1}
  }
\cs_new:Npn \__tl_map_tokens_end:w \s__tl_stop \use:n #1#2
  {
    \__tl_use_none_delimit_by_s_stop:w #2 \tl_map_break: \s__tl_stop
    #1 {#2}
    \__tl_map_tokens_end:w \s__tl_stop
  }
```

(*End of definition for* `\tl_map_tokens:nn` *and others. These functions are documented on page 119.*)

`\tl_map_variable:nNn`
`\tl_map_variable:NNn`
`\tl_map_variable:cNn`
`\__tl_map_variable:Nnn`

`\tl_map_variable:nNn` {⟨*token list*⟩} ⟨`tl var`⟩ {⟨*action*⟩} assigns ⟨`tl var`⟩ to each element and executes ⟨`action`⟩. The assignment to ⟨`tl var`⟩ is done after the quark test so that this variable does not get set to a quark.

```
\cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
  { \tl_map_tokens:nn {#1} { \__tl_map_variable:Nnn #2 {#3} } }
\cs_new_protected:Npn \__tl_map_variable:Nnn #1#2#3
  { \tl_set:Nn #1 {#3} #2 }
\cs_new_protected:Npn \tl_map_variable:NNn
  { \exp_args:No \tl_map_variable:nNn }
\cs_generate_variant:Nn \tl_map_variable:NNn { c }
```

(*End of definition for* `\tl_map_variable:nNn`, `\tl_map_variable:NNn`, *and* `\__tl_map_variable:Nnn`. *These functions are documented on page 120.*)

`\tl_map_break:`
`\tl_map_break:n`

The break statements use the general `\prg_map_break:Nn`.

```
\cs_new:Npn \tl_map_break:
  { \prg_map_break:Nn \tl_map_break: { } }
\cs_new:Npn \tl_map_break:n
  { \prg_map_break:Nn \tl_map_break: }
```

(*End of definition for* `\tl_map_break:` *and* `\tl_map_break:n`. *These functions are documented on page 120.*)

53.9 Using token lists

\tl_to_str:n
\tl_to_str:o
\tl_to_str:V
\tl_to_str:v
\tl_to_str:e

Another name for a primitive: defined in l3basics.

```
\cs_generate_variant:Nn \tl_to_str:n { o , V , v , e }
```

(*End of definition for* `\tl_to_str:n`. *This function is documented on page* 116.)

\tl_to_str:N
\tl_to_str:c

These functions return the replacement text of a token list as a string.

```
\cs_new:Npn \tl_to_str:N #1 { \__kernel_tl_to_str:w \exp_after:wN {#1} }
\cs_generate_variant:Nn \tl_to_str:N { c }
```

(*End of definition for* `\tl_to_str:N`. *This function is documented on page* 116.)

\tl_use:N
\tl_use:c

Token lists which are simply not defined give a clear TeX error here. No such luck for ones equal to `\scan_stop:` so instead a test is made and if there is an issue an error is forced.

```
\cs_new:Npn \tl_use:N #1
  {
    \tl_if_exist:NTF #1 {#1}
      {
        \msg_expandable_error:nnn
          { kernel } { bad-variable } {#1}
      }
  }
\cs_generate_variant:Nn \tl_use:N { c }
```

(*End of definition for* `\tl_use:N`. *This function is documented on page* 116.)

53.10 Working with the contents of token lists

\tl_count:n
\tl_count:V
\tl_count:v
\tl_count:e
\tl_count:o
\tl_count:N
\tl_count:c
__tl_count:n

Count number of elements within a token list or token list variable. Brace groups within the list are read as a single element. Spaces are ignored. `\__tl_count:n` grabs the element and replaces it by `+1`. The `0` ensures that it works on an empty list.

```
\cs_new:Npn \tl_count:n #1
  {
    \int_eval:n
      { 0 \tl_map_function:nN {#1} \__tl_count:n }
  }
\cs_new:Npn \tl_count:N #1
  {
    \int_eval:n
      { 0 \tl_map_function:NN #1 \__tl_count:n }
  }
\cs_new:Npn \__tl_count:n #1 { + 1 }
\cs_generate_variant:Nn \tl_count:n { V , v , e , o }
\cs_generate_variant:Nn \tl_count:N { c }
```

(*End of definition for* `\tl_count:n`, `\tl_count:N`, *and* `\__tl_count:n`. *These functions are documented on page* 117.)

\tl_count_tokens:n
__tl_act_count_normal:nN
__tl_act_count_group:nn
__tl_act_count_space:n

The token count is computed through an `\int_eval:n` construction. Each `1+` is output to the *left*, into the integer expression, and the sum is ended by the `\exp_end:` inserted by `\__tl_act_end:wn` (which is technically implemented as `\c_zero_int`). Somewhat a hack!

```
\cs_new:Npn \tl_count_tokens:n #1
  {
    \int_eval:n
      {
        \__tl_act:NNNn
          \__tl_act_count_normal:N
          \__tl_act_count_group:n
          \__tl_act_count_space:
          {#1}
      }
  }
\cs_new:Npn \__tl_act_count_normal:N #1 { 1 + }
\cs_new:Npn \__tl_act_count_space: { 1 + }
\cs_new:Npn \__tl_act_count_group:n #1 { 2 + \tl_count_tokens:n {#1} + }
```

(*End of definition for* `\tl_count_tokens:n` *and others. This function is documented on page 117.*)

\tl_reverse_items:n
__tl_reverse_items:nwNwn
__tl_reverse_items:wn

Reversal of a token list is done by taking one item at a time and putting it after `\s__tl_stop`.

```
\cs_new:Npn \tl_reverse_items:n #1
  {
    \__tl_reverse_items:nwNwn #1 ?
      \s__tl_mark \__tl_reverse_items:nwNwn
      \s__tl_mark \__tl_reverse_items:wn
      \s__tl_stop { }
  }
\cs_new:Npn \__tl_reverse_items:nwNwn #1 #2 \s__tl_mark #3 #4 \s__tl_stop #5
  {
    #3 #2
      \s__tl_mark \__tl_reverse_items:nwNwn
      \s__tl_mark \__tl_reverse_items:wn
      \s__tl_stop { {#1} #5 }
  }
\cs_new:Npn \__tl_reverse_items:wn #1 \s__tl_stop #2
  { \__kernel_exp_not:w \exp_after:wN { \use_none:nn #2 } }
```

(*End of definition for* `\tl_reverse_items:n`, `\__tl_reverse_items:nwNwn`, *and* `\__tl_reverse_items:wn`. *This function is documented on page 117.*)

\tl_trim_spaces:n
\tl_trim_spaces:V
\tl_trim_spaces:v
\tl_trim_spaces:e
\tl_trim_spaces:o
\tl_trim_spaces_apply:nN
\tl_trim_spaces_apply:oN
\tl_trim_spaces:N
\tl_trim_spaces:c
\tl_gtrim_spaces:N
\tl_gtrim_spaces:c

Trimming spaces from around the input is deferred to an internal function whose first argument is the token list to trim, augmented by an initial `\__tl_trim_mark:`, and whose second argument is a ⟨*continuation*⟩, which receives as a braced argument `\__tl_trim_mark:` ⟨*trimmed token list*⟩. The control sequence `\__tl_trim_mark:` expands to nothing in a single expansion. In the case at hand, we take `\__kernel_exp_not:w \exp_after:wN` as our continuation, so that space trimming behaves correctly within an `e`-type or `x`-type expansion.

```
\cs_new:Npn \tl_trim_spaces:n #1
  {
    \__tl_trim_spaces:nn
      { \__tl_trim_mark: #1 }
```

```
      { \__kernel_exp_not:w \exp_after:wN }
  }
\cs_generate_variant:Nn \tl_trim_spaces:n { V , v , e , o }
\cs_new:Npn \tl_trim_spaces_apply:nN #1#2
  { \__tl_trim_spaces:nn { \__tl_trim_mark: #1 } { \exp_args:No #2 } }
\cs_generate_variant:Nn \tl_trim_spaces_apply:nN { o }
\cs_new_protected:Npn \tl_trim_spaces:N #1
  { \__kernel_tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
\cs_new_protected:Npn \tl_gtrim_spaces:N #1
  { \__kernel_tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
\cs_generate_variant:Nn \tl_trim_spaces:N  { c }
\cs_generate_variant:Nn \tl_gtrim_spaces:N { c }
```

__tl_trim_spaces:nn
__tl_trim_spaces_auxi:w
__tl_trim_spaces_auxii:w
__tl_trim_spaces_auxiii:w
__tl_trim_spaces_auxiv:w
__tl_trim_mark:

Trimming spaces from around the input is done using delimited arguments and quarks, and to get spaces at odd places in the definitions, we nest those in `\__tl_tmp:w`, which then receives a single space as its argument: `#1` is ␣. Removing leading spaces is done with `\__tl_trim_spaces_auxi:w`, which loops until `\__tl_trim_mark:␣` matches the end of the token list: then `##1` is the token list and `##3` is `\__tl_trim_spaces_auxii:w`. This hands the relevant tokens to the loop `\__tl_trim_spaces_auxiii:w`, responsible for trimming trailing spaces. The end is reached when ␣ `\s__tl_nil` matches the one present in the definition of `\tl_trim_spaces:n`. Then `\__tl_trim_spaces_auxiv:w` puts the token list into a group, with a lingering `\__tl_trim_mark:` at the start (which will expand to nothing in one step of expansion), and feeds this to the ⟨*continuation*⟩.

```
\cs_set_protected:Npn \__tl_tmp:w #1
  {
    \cs_new:Npn \__tl_trim_spaces:nn ##1
      {
        \__tl_trim_spaces_auxi:w
          ##1
          \s__tl_nil
          \__tl_trim_mark: #1 { }
          \__tl_trim_mark: \__tl_trim_spaces_auxii:w
          \__tl_trim_spaces_auxiii:w
          #1 \s__tl_nil
          \__tl_trim_spaces_auxiv:w
        \s__tl_stop
      }
    \cs_new:Npn
        \__tl_trim_spaces_auxi:w ##1 \__tl_trim_mark: #1 ##2 \__tl_trim_mark: ##3
      {
        ##3
        \__tl_trim_spaces_auxi:w
        \__tl_trim_mark:
        ##2
        \__tl_trim_mark: #1 {##1}
      }
    \cs_new:Npn \__tl_trim_spaces_auxii:w
        \__tl_trim_spaces_auxi:w \__tl_trim_mark: \__tl_trim_mark: ##1
      {
        \__tl_trim_spaces_auxiii:w
        ##1
      }
    \cs_new:Npn \__tl_trim_spaces_auxiii:w ##1 #1 \s__tl_nil ##2
```

```
      {
        ##2
        ##1 \s__tl_nil
        \__tl_trim_spaces_auxiii:w
      }
    \cs_new:Npn \__tl_trim_spaces_auxiv:w ##1 \s__tl_nil ##2 \s__tl_stop ##3
      { ##3 { ##1 } }
    \cs_new:Npn \__tl_trim_mark: {}
  }
\__tl_tmp:w { ~ }
```

(*End of definition for* `\tl_trim_spaces:n` *and others. These functions are documented on page 118.*)

`\tl_sort:Nn`
`\tl_sort:cn`
`\tl_gsort:Nn`
`\tl_gsort:cn`
`\tl_sort:nN`

Implemented in l3sort.

(*End of definition for* `\tl_sort:Nn`*,* `\tl_gsort:Nn`*, and* `\tl_sort:nN`*. These functions are documented on page 124.*)

53.11 The first token from a token list

`\tl_head:N`
`\tl_head:n`
`\tl_head:V`
`\tl_head:v`
`\tl_head:f`
`\__tl_head_auxi:nw`
`\__tl_head_auxii:n`
`\tl_head:w`
`\__tl_tl_head:w`
`\tl_tail:N`
`\tl_tail:n`
`\tl_tail:V`
`\tl_tail:v`
`\tl_tail:f`

Finding the head of a token list expandably always strips braces, which is fine as this is consistent with for example mapping over a list. The empty brace groups in `\tl_head:n` ensure that a blank argument gives an empty result. The result is returned within the `\unexpanded` primitive. The approach here is to use `\if_false:` to allow us to use } as the closing delimiter: this is the only safe choice, as any other token would not be able to parse it's own code. More detail in http://tex.stackexchange.com/a/70168.

```
\cs_new:Npn \tl_head:n #1
  {
    \__kernel_exp_not:w \tex_expanded:D
      { { \if_false: { \fi: \__tl_head_aux:n #1 { } } } }
  }
\cs_new:Npn \__tl_head_aux:n #1
  {
    \__kernel_exp_not:w {#1}
    \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
  }
\cs_generate_variant:Nn \tl_head:n { V , v , f }
\cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
\cs_new:Npn \__tl_tl_head:w #1#2 \s__tl_stop {#1}
\cs_new:Npn \tl_head:N { \exp_args:No \tl_head:n }
```

To correctly leave the tail of a token list, it's important *not* to absorb any of the tail part as an argument. For example, the simple definition

```
\cs_new:Npn \tl_tail:n #1 { \tl_tail:w #1 \q_stop }
\cs_new:Npn \tl_tail:w #1#2 \q_stop
```

would give the wrong result for `\tl_tail:n { a { bc } }` (the braces would be stripped). Thus the only safe way to proceed is to first check that there is an item to grab (*i.e.* that the argument is not blank) and assuming there is to dispose of the first item. As with `\tl_head:n`, the result is protected from further expansion by `\unexpanded`. While we could optimise the test here, this would leave some tokens "banned" in the input, which we do not have with this definition.

```
\exp_args:Nno \use:n { \cs_new:Npn \tl_tail:n #1 }
  {
    \exp_after:wN \__kernel_exp_not:w
      \tl_if_blank:nTF {#1}
        { { } }
        { \exp_after:wN { \use_none:n #1 } }
  }
\cs_generate_variant:Nn \tl_tail:n { V , v , f }
\cs_new:Npn \tl_tail:N { \exp_args:No \tl_tail:n }
```

(*End of definition for* `\tl_head:N` *and others. These functions are documented on page 121.*)

`\tl_if_head_eq_meaning_p:nN`
`\tl_if_head_eq_meaning_p:VN`
`\tl_if_head_eq_meaning_p:eN`
`\tl_if_head_eq_meaning:nNTF`
`\tl_if_head_eq_meaning:VNTF`
`\tl_if_head_eq_meaning:eNTF`
`\tl_if_head_eq_charcode_p:nN`
`\tl_if_head_eq_charcode_p:VN`
`\tl_if_head_eq_charcode_p:eN`
`\tl_if_head_eq_charcode_p:fN`
`\tl_if_head_eq_charcode:nNTF`
`\tl_if_head_eq_charcode:VNTF`
`\tl_if_head_eq_charcode:eNTF`
`\tl_if_head_eq_charcode:fNTF`
`\tl_if_head_eq_catcode_p:nN`
`\tl_if_head_eq_catcode_p:VN`
`\tl_if_head_eq_catcode_p:eN`
`\tl_if_head_eq_catcode_p:oN`
`\tl_if_head_eq_catcode:nNTF`
`\tl_if_head_eq_catcode:VNTF`
`\tl_if_head_eq_catcode:eNTF`
`\tl_if_head_eq_catcode:oNTF`
`\__tl_head_exp_not:w`
`\__tl_if_head_eq_empty_arg:w`

Accessing the first token of a token list is tricky in three cases: when it has category code 1 (begin-group token), when it is an explicit space, with category code 10 and character code 32, or when the token list is empty (obviously).

Forgetting temporarily about this issue we would use the following test in `\tl_if_head_eq_charcode:nN`. Here, `\tl_head:w` yields the first token of the token list, then passed to `\exp_not:N`.

```
\if_charcode:w
    \exp_after:wN \exp_not:N \tl_head:w #1 \q_nil \q_stop
    \exp_not:N #2
```

The two first special cases are detected by testing if the token list starts with an N-type token (the extra `?` sends empty token lists to the `true` branch of this test). In those cases, the first token is a character, and since we only care about its character code, we can use `\str_head:n` to access it (this works even if it is a space character). An empty argument results in `\tl_head:w` leaving two token: `^` and `\__tl_if_head_eq_empty_arg:w` which will result in the `\if_charcode:w` test being false and remove `\exp_not:N` and `#2`.

```
\prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
  {
    \if_charcode:w
        \tl_if_head_is_N_type:nTF { #1 ? }
          { \__tl_head_exp_not:w #1 { ^ \__tl_if_head_eq_empty_arg:w } \s__tl_stop }
          { \str_head:n {#1} }
        \exp_not:N #2
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_generate_conditional_variant:Nnn \tl_if_head_eq_charcode:nN
  { V , e , f } { p , TF , T , F }
```

For `\tl_if_head_eq_catcode:nN`, again we detect special cases with a `\tl_if_head_is_N_type:n`. Then we need to test if the first token is a begin-group token or an explicit space token, and produce the relevant token, either `\c_group_begin_token` or `\c_space_token`. Again, for an empty argument, a hack is used, removing the token given by the user and leaving two tokens in the input stream which will make the `\if_catcode:w` test return `false`.

```
\prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
  {
    \if_catcode:w
        \tl_if_head_is_N_type:nTF { #1 ? }
```

```
          { \__tl_head_exp_not:w #1 { ^ \__tl_if_head_eq_empty_arg:w } \s__tl_stop }
          {
            \tl_if_head_is_group:nTF {#1}
              \c_group_begin_token
              \c_space_token
          }
        \exp_not:N #2
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_generate_conditional_variant:Nnn \tl_if_head_eq_catcode:nN
  { V , e , o } { p , TF , T , F }
```

For `\tl_if_head_eq_meaning:nN`, again, detect special cases. In the normal case, use `\tl_head:w`, with no `\exp_not:N` this time, since `\if_meaning:w` causes no expansion. With an empty argument, the test is `true`, and `\use_none:nnn` removes `#2` and `\prg_return_true:` and `\else:` (it is safe this way here as in this case `\prg_new_conditional:Npnn` didn't optimize these two away). In the special cases, we know that the first token is a character, hence `\if_charcode:w` and `\if_catcode:w` together are enough. We combine them in some order, hopefully faster than the reverse. Tests are not nested because the arguments may contain unmatched primitive conditionals.

```
\prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
  {
    \tl_if_head_is_N_type:nTF { #1 ? }
      \__tl_if_head_eq_meaning_normal:nN
      \__tl_if_head_eq_meaning_special:nN
    {#1} #2
  }
\prg_generate_conditional_variant:Nnn \tl_if_head_eq_meaning:nN
  { V , e } { p , TF , T , F }
\cs_new:Npn \__tl_if_head_eq_meaning_normal:nN #1 #2
  {
    \exp_after:wN \if_meaning:w
        \__tl_tl_head:w #1 { ?? \use_none:nnn } \s__tl_stop #2
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \__tl_if_head_eq_meaning_special:nN #1 #2
  {
    \if_charcode:w \str_head:n {#1} \exp_not:N #2
      \exp_after:wN \use_ii:nn
    \else:
      \prg_return_false:
    \fi:
    \use_none:n
    {
      \if_catcode:w \exp_not:N #2
                    \tl_if_head_is_group:nTF {#1}
                      { \c_group_begin_token }
                      { \c_space_token }
```

```
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    }
  }
```

Both `\tl_if_head_eq_charcode:nN` and `\tl_if_head_eq_catcode:nN` will need to get the first token of their argument and apply `\exp_not:N` to it. `\__tl_head_exp_not:w` does exactly that.

```
\cs_new:Npn \__tl_head_exp_not:w #1 #2 \s__tl_stop
  { \exp_not:N #1 }
```

If the argument of `\tl_if_head_eq_charcode:nN` and `\tl_if_head_eq_catcode:nN` was empty `\__tl_if_head_eq_empty_arg:w` will be left in the input stream. This macro has to remove `\exp_not:N` and the following token from the input stream to make sure no unbalanced if-construct is created and leave tokens there which make the two tests return `false`.

```
\cs_new:Npn \__tl_if_head_eq_empty_arg:w \exp_not:N #1
  { ? }
```

(*End of definition for* `\tl_if_head_eq_meaning:nNTF` *and others. These functions are documented on page 115.*)

`\tl_if_head_is_N_type_p:n`
`\tl_if_head_is_N_type:nTF`
`\__tl_if_head_is_N_type_auxi:w`
`\__tl_if_head_is_N_type_auxii:n`

A token list can be empty, can start with an explicit space character (catcode 10 and charcode 32), can start with a begin-group token (catcode 1), or start with an N-type argument. In the first two cases, and when `#1~` starts with `{}~`, `\__tl_if_head_is_N_type_auxi:w` receives an empty argument hence produces `f` and removes everything before the first `\scan_stop:`. In the third case (except when `#1~` starts with `{}~`), the second auxiliary removes the first copy of `#1` that was used for the space test, then expands `\token_to_str:N` which hits the leading begin-group token, leaving a single closing brace to be compared with `\scan_stop:`. In the last case, `\token_to_str:N` does not change the brace balance so that only `\scan_stop: \scan_stop:` remain, making the character code test true. One cannot optimize by moving one of the `\scan_stop:` to the beginning: if `#1` contains primitive conditionals, all of its occurrences must be dealt with before the `\if:w` tries to skip the `true` branch of the conditional.

```
\prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
  {
    \if:w
        \if_false: { \fi: \__tl_if_head_is_N_type_auxi:w #1 ~ }
        { \exp_after:wN { \token_to_str:N #1 } }
        \scan_stop: \scan_stop:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\exp_args:Nno \use:n { \cs_new:Npn \__tl_if_head_is_N_type_auxi:w #1 ~ }
  {
    \tl_if_empty:nTF {#1}
      { f \exp_after:wN \use_none:nn }
      { \exp_after:wN \__tl_if_head_is_N_type_auxii:n }
    \exp_after:wN { \if_false: } \fi:
  }
```

```
\cs_new:Npn \__tl_if_head_is_N_type_auxii:n #1
  { \exp_after:wN \use_none:n \exp_after:wN }
```

(*End of definition for* `\tl_if_head_is_N_type:nTF`*,* `\__tl_if_head_is_N_type_auxi:w`*, and* `\__tl_if_head_is_N_type_auxii:n`*. This function is documented on page 115.*)

`\tl_if_head_is_group_p:n`
`\tl_if_head_is_group:nTF`
`\__tl_if_head_is_group_fi_false:w`

Pass the first token of `#1` through `\token_to_str:N`, then check for the brace balance. The extra `?` caters for an empty argument. This could be made faster, but we need all brace tricks to happen in one step of expansion, keeping the token list brace balanced at all times.

```
\prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
  {
    \if:w
        \exp_after:wN \use_none:n
          \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
        \scan_stop: \scan_stop:
      \__tl_if_head_is_group_fi_false:w
    \fi:
    \if_true:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \__tl_if_head_is_group_fi_false:w \fi: \if_true: { \fi: \if_false: }
```

(*End of definition for* `\tl_if_head_is_group:nTF` *and* `\__tl_if_head_is_group_fi_false:w`*. This function is documented on page 115.*)

`\tl_if_head_is_space_p:n`
`\tl_if_head_is_space:nTF`
`\__tl_if_head_is_space:w`

The auxiliary's argument is all that is before the first explicit space in `\prg_do_nothing:#1?~`. If that is a single `\prg_do_nothing:` the test yields `true`. Otherwise, that is more than one token, and the test yields `false`. The work is done within braces (with an `\if_false: { \fi: ... }` construction) both to hide potential alignment tab characters from TeX in a table, and to allow for removing what remains of the token list after its first space. The use of `\if:w` ensures that the result of a single step of expansion directly yields a balanced token list (no trailing closing brace).

```
\prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
  {
    \if:w
        \if_false: { \fi: \__tl_if_head_is_space:w \prg_do_nothing: #1 ? ~ }
        \scan_stop: \scan_stop:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\exp_args:Nno \use:n { \cs_new:Npn \__tl_if_head_is_space:w #1 ~ }
  {
    \__tl_if_empty_if:o {#1} \else: f \fi:
    \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
  }
```

(*End of definition for* `\tl_if_head_is_space:nTF` *and* `\__tl_if_head_is_space:w`*. This function is documented on page 116.*)

53.12 Token by token changes

\s__tl_act_stop

The __tl_act_... functions may be applied to any token list. Hence, we use a private quark, to allow any token, even quarks, in the token list. Only \s__tl_act_stop may not appear in the token lists manipulated by __tl_act:NNNn functions.

```
\scan_new:N \s__tl_act_stop
```

(*End of definition for* \s__tl_act_stop.)

__tl_act:NNNn
__tl_act_output:n
__tl_act_reverse_output:n
__tl_act_loop:w
__tl_act_normal:NwNNN
__tl_act_group:nwNNN
__tl_act_space:wwNNN
__tl_act_end:wn
__tl_act_if_head_is_space:nTF
__tl_act_if_head_is_space:w
__tl_act_if_head_is_space_true:w
__tl_use_none_delimit_by_q_act_stop:w

To help control the expansion, __tl_act:NNNn should always be preceded by \exp:w and ends by producing \exp_end: once the result has been obtained. This way no internal token of it can be accidentally end up in the input stream. Because \s__tl_act_stop can't appear without braces around it in the argument #1 of __tl_act_loop:w, we can use this marker to set up a fast test for leading spaces.

```
\cs_set_protected:Npn \__tl_tmp:w #1
  {
    \cs_new:Npn \__tl_act_if_head_is_space:nTF ##1
      {
        \__tl_act_if_head_is_space:w
          \s__tl_act_stop ##1 \s__tl_act_stop \__tl_act_if_head_is_space_true:w
          \s__tl_act_stop #1  \s__tl_act_stop \use_ii:nn
      }
    \cs_new:Npn \__tl_act_if_head_is_space:w
        ##1 \s__tl_act_stop #1 ##2 \s__tl_act_stop
      {}
    \cs_new:Npn \__tl_act_if_head_is_space_true:w
        \s__tl_act_stop #1 \s__tl_act_stop \use_ii:nn ##1 ##2
      {##1}
  }
\__tl_tmp:w { ~ }
```

(We expand the definition __tl_act_if_head_is_space:nTF when setting up __tl_act_loop:w, so we can then undefine the auxiliary.) In the loop, we check how the token list begins and act accordingly. In the "group" case, we may have reached \s__tl_act_stop, the end of the list. Then leave \exp_end: and the result in the input stream, to terminate the expansion of \exp:w. Otherwise, apply the relevant function to the "arguments", #3 and to the head of the token list. Then repeat the loop. The scheme is the same if the token list starts with an N-type or with a space, making sure that __tl_act_space:wwNNN gobbles the space.

```
\exp_args:Nne \use:n { \cs_new:Npn \__tl_act_loop:w #1 \s__tl_act_stop }
  {
    \exp_not:o { \__tl_act_if_head_is_space:nTF {#1} }
      \exp_not:N \__tl_act_space:wwNNN
      {
        \exp_not:o { \tl_if_head_is_group:nTF {#1} }
          \exp_not:N \__tl_act_group:nwNNN
          \exp_not:N \__tl_act_normal:NwNNN
      }
    \exp_not:n {#1} \s__tl_act_stop
  }
\cs_undefine:N \__tl_act_if_head_is_space:nTF
\cs_new:Npn \__tl_act_normal:NwNNN #1 #2 \s__tl_act_stop #3
  {
```

```
    #3 #1
    \__tl_act_loop:w #2 \s__tl_act_stop
    #3
  }
\cs_new:Npn \__tl_use_none_delimit_by_s_act_stop:w #1 \s__tl_act_stop { }
\cs_new:Npn \__tl_act_end:wn #1 \__tl_act_result:n #2
  { \group_align_safe_end: \exp_end: #2 }
\cs_new:Npn \__tl_act_group:nwNNN #1 #2 \s__tl_act_stop #3#4#5
  {
    \__tl_use_none_delimit_by_s_act_stop:w #1 \__tl_act_end:wn \s__tl_act_stop
    #5 {#1}
    \__tl_act_loop:w #2 \s__tl_act_stop
    #3 #4 #5
  }
\exp_last_unbraced:NNo
  \cs_new:Npn \__tl_act_space:wwNNN \c_space_tl #1 \s__tl_act_stop #2#3
  {
    #3
    \__tl_act_loop:w #1 \s__tl_act_stop
    #2 #3
  }
```

`\__tl_act:NNNn` loops over tokens, groups, and spaces in `#4`. `{\s_@@_act_stop}` serves as the end of token list marker, the `?` after it avoids losing outer braces. The result is stored as an argument for the dummy function `\__tl_act_result:n`.

```
\cs_new:Npn \__tl_act:NNNn #1#2#3#4
  {
    \group_align_safe_begin:
    \__tl_act_loop:w #4 { \s__tl_act_stop } ? \s__tl_act_stop
    #1 #3 #2
    \__tl_act_result:n { }
  }
```

Typically, the output is done to the right of what was already output, using `\__tl_act_output:n`, but for the `\__tl_act_reverse` functions, it should be done to the left.

```
\cs_new:Npn \__tl_act_output:n #1 #2 \__tl_act_result:n #3
  { #2 \__tl_act_result:n { #3 #1 } }
\cs_new:Npn \__tl_act_reverse_output:n #1 #2 \__tl_act_result:n #3
  { #2 \__tl_act_result:n { #1 #3 } }
```

(*End of definition for* `\__tl_act:NNNn` *and others.*)

`\tl_reverse:n`
`\tl_reverse:o`
`\tl_reverse:V`
`\tl_reverse:f`
`\tl_reverse:e`
`\__tl_reverse_normal:nN`
`\__tl_reverse_group_preserve:nn`
`\__tl_reverse_space:n`

The goal here is to reverse without losing spaces nor braces. This is done using the general internal function `\__tl_act:NNNn`. Spaces and "normal" tokens are output on the left of the current output. Grouped tokens are output to the left but without any reversal within the group.

```
\cs_new:Npn \tl_reverse:n #1
  {
    \__kernel_exp_not:w \exp_after:wN
      {
        \exp:w
        \__tl_act:NNNn
          \__tl_reverse_normal:N
          \__tl_reverse_group_preserve:n
          \__tl_reverse_space:
```

```
          {#1}
      }
  }
\cs_generate_variant:Nn \tl_reverse:n { o , V , f , e }
\cs_new:Npn \__tl_reverse_normal:N
  { \__tl_act_reverse_output:n }
\cs_new:Npn \__tl_reverse_group_preserve:n #1
  { \__tl_act_reverse_output:n { {#1} } }
\cs_new:Npn \__tl_reverse_space:
  { \__tl_act_reverse_output:n { ~ } }
```

(*End of definition for* `\tl_reverse:n` *and others. This function is documented on page 117.*)

`\tl_reverse:N` `\tl_reverse:c` `\tl_greverse:N` `\tl_greverse:c`

This reverses the list, leaving `\exp_stop_f:` in front, which stops the f-expansion.

```
\cs_new_protected:Npn \tl_reverse:N #1
  { \__kernel_tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
\cs_new_protected:Npn \tl_greverse:N #1
  { \__kernel_tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
\cs_generate_variant:Nn \tl_reverse:N  { c }
\cs_generate_variant:Nn \tl_greverse:N { c }
```

(*End of definition for* `\tl_reverse:N` *and* `\tl_greverse:N`*. These functions are documented on page 117.*)

53.13 Using a single item

`\tl_item:nn` `\tl_item:Nn` `\tl_item:cn` `\__tl_item_aux:nn` `\__tl_item:nn`

The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then `\__tl_if_recursion_tail_break:nN` terminates the loop, and returns nothing at all.

```
\cs_new:Npn \tl_item:nn #1#2
  {
    \exp_args:Nf \__tl_item:nn
      { \exp_args:Nf \__tl_item_aux:nn { \int_eval:n {#2} } {#1} }
    #1
    \q__tl_recursion_tail
    \prg_break_point:
  }
\cs_new:Npn \__tl_item_aux:nn #1#2
  {
    \int_compare:nNnTF {#1} < 0
      { \int_eval:n { \tl_count:n {#2} + 1 + #1 } }
      {#1}
  }
\cs_new:Npn \__tl_item:nn #1#2
  {
    \__tl_if_recursion_tail_break:nN {#2} \prg_break:
    \int_compare:nNnTF {#1} = 1
      { \prg_break:n { \exp_not:n {#2} } }
      { \exp_args:Nf \__tl_item:nn { \int_eval:n { #1 - 1 } } }
  }
\cs_new:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
\cs_generate_variant:Nn \tl_item:Nn { c }
```

(*End of definition for* `\tl_item:nn` *and others. These functions are documented on page 122.*)

\tl_rand_item:n
\tl_rand_item:N
\tl_rand_item:c

Importantly \tl_item:nn only evaluates its argument once.

```
\cs_new:Npn \tl_rand_item:n #1
  {
    \tl_if_blank:nF {#1}
      { \tl_item:nn {#1} { \int_rand:nn { 1 } { \tl_count:n {#1} } } }
  }
\cs_new:Npn \tl_rand_item:N { \exp_args:No \tl_rand_item:n }
\cs_generate_variant:Nn \tl_rand_item:N { c }
```

(*End of definition for* \tl_rand_item:n *and* \tl_rand_item:N*. These functions are documented on page 122.*)

\tl_range:Nnn
\tl_range:cnn
\tl_range:nnn
__tl_range:Nnnn
__tl_range:nnnNn
__tl_range:nnNn
__tl_range_skip:w
__tl_range:w
__tl_range_skip_spaces:n
__tl_range_collect:nn
__tl_range_collect:ff
__tl_range_collect_space:nw
__tl_range_collect_N:nN
__tl_range_collect_group:nN

To avoid checking for the end of the token list at every step, start by counting the number l of items and "normalizing" the bounds, namely clamping them to the interval $[0, l]$ and dealing with negative indices. More precisely, __tl_range_items:nnNn receives the number of items to skip at the beginning of the token list, the index of the last item to keep, a function which is either __tl_range:w or the token list itself. If nothing should be kept, leave {}: this stops the f-expansion of \tl_head:f and that function produces an empty result. Otherwise, repeatedly call __tl_range_skip:w to delete #1 items from the input stream (the extra brace group avoids an off-by-one shift). For the braced version __tl_range_braced:w sets up __tl_range_collect_braced:w which stores items one by one in an argument after the semicolon. Depending on the first token of the tail, either just move it (if it is a space) or also decrement the number of items left to find. Eventually, the result is a brace group followed by the rest of the token list, and \tl_head:f cleans up and gives the result in \exp_not:n.

```
\cs_new:Npn \tl_range:Nnn { \exp_args:No \tl_range:nnn }
\cs_generate_variant:Nn \tl_range:Nnn { c }
\cs_new:Npn \tl_range:nnn { \__tl_range:Nnnn \__tl_range:w }
\cs_new:Npn \__tl_range:Nnnn #1#2#3#4
  {
    \tl_head:f
      {
        \exp_args:Nf \__tl_range:nnnNn
          { \tl_count:n {#2} } {#3} {#4} #1 {#2}
      }
  }
\cs_new:Npn \__tl_range:nnnNn #1#2#3
  {
    \exp_args:Nff \__tl_range:nnNn
      {
        \exp_args:Nf \__tl_range_normalize:nn
          { \int_eval:n { #2 - 1 } } {#1}
      }
      {
        \exp_args:Nf \__tl_range_normalize:nn
          { \int_eval:n {#3} } {#1}
      }
  }
\cs_new:Npn \__tl_range:nnNn #1#2#3#4
  {
    \if_int_compare:w #2 > #1 \exp_stop_f: \else:
      \exp_after:wN { \exp_after:wN }
    \fi:
```

```
    \exp_after:wN #3
    \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ;
    \exp_after:wN { \exp:w \__tl_range_skip:w #1 ; { } #4 }
  }
\cs_new:Npn \__tl_range_skip:w #1 ; #2
  {
    \if_int_compare:w #1 > \c_zero_int
      \exp_after:wN \__tl_range_skip:w
      \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
    \else:
      \exp_after:wN \exp_end:
    \fi:
  }
\cs_new:Npn \__tl_range:w #1 ; #2
  {
    \exp_args:Nf \__tl_range_collect:nn
      { \__tl_range_skip_spaces:n {#2} } {#1}
  }
\cs_new:Npn \__tl_range_skip_spaces:n #1
  {
    \tl_if_head_is_space:nTF {#1}
      { \exp_args:Nf \__tl_range_skip_spaces:n {#1} }
      { { } #1 }
  }
\cs_new:Npn \__tl_range_collect:nn #1#2
  {
    \int_compare:nNnTF {#2} = 0
      {#1}
      {
        \exp_args:No \tl_if_head_is_space:nTF { \use_none:n #1 }
          {
            \exp_args:Nf \__tl_range_collect:nn
              { \__tl_range_collect_space:nw #1 }
              {#2}
          }
          {
            \__tl_range_collect:ff
              {
                \exp_args:No \tl_if_head_is_N_type:nTF { \use_none:n #1 }
                  { \__tl_range_collect_N:nN }
                  { \__tl_range_collect_group:nn }
                #1
              }
              { \int_eval:n { #2 - 1 } }
          }
      }
  }
\cs_new:Npn \__tl_range_collect_space:nw #1 ~ { { #1 ~ } }
\cs_new:Npn \__tl_range_collect_N:nN #1#2 { { #1 #2 } }
\cs_new:Npn \__tl_range_collect_group:nn #1#2 { { #1 {#2} } }
\cs_generate_variant:Nn \__tl_range_collect:nn { ff }
```

(*End of definition for* `\tl_range:Nnn` *and others. These functions are documented on page 123.*)

`\__tl_range_normalize:nn` This function converts an ⟨*index*⟩ argument into an explicit position in the token list (a result of 0 denoting "out of bounds"). Expects two explicit integer arguments: the ⟨*index*⟩ #1 and the string count #2. If #1 is negative, replace it by $#1 + #2 + 1$, then limit to the range $[0, #2]$.

```
\cs_new:Npn \__tl_range_normalize:nn #1#2
  {
    \int_eval:n
      {
        \if_int_compare:w #1 < \c_zero_int
          \if_int_compare:w #1 < -#2 \exp_stop_f:
            0
          \else:
            #1 + #2 + 1
          \fi:
        \else:
          \if_int_compare:w #1 < #2 \exp_stop_f:
            #1
          \else:
            #2
          \fi:
        \fi:
      }
  }
```

(*End of definition for* `\__tl_range_normalize:nn`.)

53.14 Viewing token lists

`\tl_show:N`
`\tl_show:c`
`\tl_log:N`
`\tl_log:c`
`\__tl_show:NN`

Showing token list variables is done after checking that the variable is defined (see `\__kernel_register_show:N`).

```
\cs_new_protected:Npn \tl_show:N { \__tl_show:NN \tl_show:n }
\cs_generate_variant:Nn \tl_show:N { c }
\cs_new_protected:Npn \tl_log:N { \__tl_show:NN \tl_log:n }
\cs_generate_variant:Nn \tl_log:N { c }
\cs_new_protected:Npn \__tl_show:NN #1#2
  {
    \__kernel_chk_defined:NT #2
      {
        \exp_args:Nf \tl_if_empty:nTF
          { \cs_prefix_spec:N #2 \cs_parameter_spec:N #2 }
          {
            \exp_args:Ne #1
              { \token_to_str:N #2 = \__kernel_exp_not:w \exp_after:wN {#2} }
          }
          {
            \msg_error:nneee { kernel } { bad-type }
              { \token_to_str:N #2 } { \token_to_meaning:N #2 } { tl }
          }
      }
  }
```

(*End of definition for* `\tl_show:N`*,* `\tl_log:N`*, and* `\__tl_show:NN`*. These functions are documented on page 118.*)

\tl_show:n
\tl_show:e
\tl_show:x
__tl_show:n
__tl_show:w

Many `show` functions are based on `\tl_show:n`. The argument of `\tl_show:n` is line-wrapped using `\iow_wrap:nnnN` but with a leading `>~` and trailing period, both removed before passing the wrapped text to the `\showtokens` primitive. This primitive shows the result with a leading `>~` and trailing period.

The token list `\l__tl_internal_a_tl` containing the result of all these manipulations is displayed to the terminal using `\tex_showtokens:D` and an odd `\exp_after:wN` which expand the closing brace to improve the output slightly. The calls to `\__kernel_iow_with:Nnn` ensure that the `\newlinechar` is set to 10 so that the `\iow_newline:` inserted by the line-wrapping code are correctly recognized by TeX, and that `\errorcontextlines` is -1 to avoid printing irrelevant context.

```
\cs_new_protected:Npn \tl_show:n #1
  { \iow_wrap:nnnN { >~ \tl_to_str:n {#1} . } { } { } \__tl_show:n }
\cs_generate_variant:Nn \tl_show:n { e , x }
\cs_new_protected:Npn \__tl_show:n #1
  {
    \tl_set:Nf \l__tl_internal_a_tl { \__tl_show:w #1 \s__tl_stop }
    \__kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
      {
        \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
          {
            \tex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
              { \exp_after:wN \l__tl_internal_a_tl }
          }
      }
  }
\cs_new:Npn \__tl_show:w #1 > #2 . \s__tl_stop {#2}
```

(*End of definition for* `\tl_show:n` *,* `\__tl_show:n` *, and* `\__tl_show:w`*. This function is documented on page 118.*)

\tl_log:n
\tl_log:e
\tl_log:x

Logging is much easier, simply line-wrap. The `>~` and trailing period is there to match the output of `\tl_show:n`.

```
\cs_new_protected:Npn \tl_log:n #1
  { \iow_wrap:nnnN { > ~ \tl_to_str:n {#1} . } { } { } \iow_log:n }
\cs_generate_variant:Nn \tl_log:n { e , x }
```

(*End of definition for* `\tl_log:n`*. This function is documented on page 118.*)

__kernel_chk_tl_type:NnnT

Helper for checking that `#1` has the correct internal structure to be of a certain type. Make sure that it is defined and that it is a token list, namely a macro with no `\long` nor `\protected` prefix. Then compare `#1` to an attempt at reconstructing a valid structure of the given type using `#2` (see implementation of `\seq_show:N` for instance). If that is successful run the requested code `#4`.

```
\cs_new_protected:Npn \__kernel_chk_tl_type:NnnT #1#2#3#4
  {
    \__kernel_chk_defined:NT #1
      {
        \exp_args:Nf \tl_if_empty:nTF
          { \cs_prefix_spec:N #1 \cs_parameter_spec:N #1 }
          {
            \tl_set:Ne \l__tl_internal_a_tl {#3}
            \tl_if_eq:NNTF #1 \l__tl_internal_a_tl
              {#4}
```

```
              {
                \msg_error:nneeeee { kernel } { bad-type }
                  { \token_to_str:N #1 } { \tl_to_str:N #1 }
                  {#2} { \tl_to_str:N \l__tl_internal_a_tl }
              }
          }
          {
            \msg_error:nneee { kernel } { bad-type }
              { \token_to_str:N #1 } { \token_to_meaning:N #1 } {#2}
          }
      }
  }
```

(*End of definition for* `\__kernel_chk_tl_type:NnnT`*.*)

53.15 Internal scan marks

\s__tl_nil
\s__tl_mark
\s__tl_stop

Internal scan marks. These are defined here at the end because the code for `\scan_new:N` depends on some l3tl functions.

```
\scan_new:N \s__tl_nil
\scan_new:N \s__tl_mark
\scan_new:N \s__tl_stop
```

(*End of definition for* `\s__tl_nil`*,* `\s__tl_mark`*, and* `\s__tl_stop`*.*)

53.16 Scratch token lists

\g_tmpa_tl
\g_tmpb_tl

Global temporary token list variables. They are supposed to be set and used immediately, with no delay between the definition and the use because you can't count on other macros not to redefine them from under you.

```
\tl_new:N \g_tmpa_tl
\tl_new:N \g_tmpb_tl
```

(*End of definition for* `\g_tmpa_tl` *and* `\g_tmpb_tl`*. These variables are documented on page 127.*)

\l_tmpa_tl
\l_tmpb_tl

These are local temporary token list variables. Be sure not to assume that the value you put into them will survive for long—see discussion above.

```
\tl_new:N \l_tmpa_tl
\tl_new:N \l_tmpb_tl
```

(*End of definition for* `\l_tmpa_tl` *and* `\l_tmpb_tl`*. These variables are documented on page 127.*)

We finally clean up a temporary control sequence that we have used at various points to set up some definitions.

```
\cs_undefine:N \__tl_tmp:w
⟨/package⟩
```

Chapter 54

l3tl-build implementation

```
⟨*package⟩
⟨@@=tl⟩
```

Between \tl_build_begin:N ⟨*tl var*⟩ and \tl_build_end:N ⟨*tl var*⟩, the ⟨*tl var*⟩ has the structure

\exp_end: ... \exp_end: __tl_build_last:NNn ⟨*assignment*⟩ ⟨*next tl*⟩ {⟨*left*⟩} ⟨*right*⟩

where ⟨*right*⟩ is not braced. The "data" it represents is ⟨*left*⟩ followed by the "data" of ⟨*next tl*⟩ followed by ⟨*right*⟩. The ⟨*next tl*⟩ is a token list variable whose name is that of ⟨*tl var*⟩ followed by '. There are between 0 and 4 \exp_end: to keep track of when ⟨*left*⟩ and ⟨*right*⟩ should be put into the ⟨*next tl*⟩. The ⟨*assignment*⟩ is \cs_set_nopar:Npe if the variable is local, and \cs_gset_nopar:Npe if it is global.

\tl_build_begin:N
\tl_build_gbegin:N
__tl_build_begin:NN
__tl_build_begin:NNN

First construct the ⟨*next tl*⟩: using a prime here conflicts with the usual expl3 convention but we need a name that can be derived from #1 without any external data such as a counter. Empty that ⟨*next tl*⟩ and setup the structure. The local and global versions only differ by a single function \cs_(g)set_nopar:Npe used for all assignments: this is important because only that function is stored in the ⟨*tl var*⟩ and ⟨*next tl*⟩ for subsequent assignments. In principle __tl_build_begin:NNN could use \tl_(g)clear_-new:N to empty #1 and make sure it is defined, but logging the definition does not seem useful so we just do #3 #1 {} to clear it locally or globally as appropriate.

```
\cs_new_protected:Npn \tl_build_begin:N #1
  { \__tl_build_begin:NN \cs_set_nopar:Npe #1 }
\cs_new_protected:Npn \tl_build_gbegin:N #1
  { \__tl_build_begin:NN \cs_gset_nopar:Npe #1 }
\cs_new_protected:Npn \__tl_build_begin:NN #1#2
  { \exp_args:Nc \__tl_build_begin:NNN { \cs_to_str:N #2 ' } #2 #1 }
\cs_new_protected:Npn \__tl_build_begin:NNN #1#2#3
  {
    #3 #1 { }
    #3 #2
      {
        \exp_not:n { \exp_end: \exp_end: \exp_end: \exp_end: }
        \exp_not:n { \__tl_build_last:NNn #3 #1 { } }
      }
  }
```

(*End of definition for `\tl_build_begin:N` and others. These functions are documented on page 128.*)

`\tl_build_put_right:Nn`
`\tl_build_put_right:Ne`
`\tl_build_put_right:Nx`
`\tl_build_gput_right:Nn`
`\tl_build_gput_right:Ne`
`\tl_build_gput_right:Nx`
`\__tl_build_last:NNn`
`\__tl_build_put:nn`
`\__tl_build_put:nw`

Similar to `\tl_put_right:Nn`, but apply `\exp:w` to `#1`. Most of the time this just removes one `\exp_end:`. When there are none left, `\__tl_build_last:NNn` is expanded instead. It resets the definition of the ⟨*tl var*⟩ by ending the `\exp_not:n` and the definition early. Then it makes sure the ⟨*next tl*⟩ (its argument `#1`) is set-up and starts a new definition. Then `\__tl_build_put:nn` and `\__tl_build_put:nw` place the ⟨*left*⟩ part of the original ⟨*tl var*⟩ as appropriate for the definition of the ⟨*next tl*⟩ (the ⟨*right*⟩ part is left in the right place without ever becoming a macro argument). We use `\exp_after:wN` rather than some `\exp_args:No` to avoid reading arguments that are likely very long token lists. We use `\cs_(g)set_nopar:Npe` rather than `\tl_(g)set:Ne` partly for the same reason and partly because the assignments are interrupted by brace tricks, which implies that the assignment does not simply set the token list to an e-expansion of the second argument.

```
\cs_new_protected:Npn \tl_build_put_right:Nn #1#2
  {
    \cs_set_nopar:Npe #1
      { \__kernel_exp_not:w \exp_after:wN { \exp:w #1 #2 } }
  }
\cs_generate_variant:Nn \tl_build_put_right:Nn { Ne , Nx }
\cs_new_protected:Npn \tl_build_gput_right:Nn #1#2
  {
    \cs_gset_nopar:Npe #1
      { \__kernel_exp_not:w \exp_after:wN { \exp:w #1 #2 } }
  }
\cs_generate_variant:Nn \tl_build_gput_right:Nn { Ne , Nx }
\cs_new_protected:Npn \__tl_build_last:NNn #1#2
  {
    \if_false: { { \fi:
          \exp_end: \exp_end: \exp_end: \exp_end: \exp_end:
          \__tl_build_last:NNn #1 #2 { }
        }
      }
    \if_meaning:w \c_empty_tl #2
      \__tl_build_begin:NN #1 #2
    \fi:
    #1 #2
      {
        \__kernel_exp_not:w \exp_after:wN
          {
            \exp:w \if_false: } } \fi:
            \exp_after:wN \__tl_build_put:nn \exp_after:wN {#2}
  }
\cs_new_protected:Npn \__tl_build_put:nn #1#2 { \__tl_build_put:nw {#2} #1 }
\cs_new_protected:Npn \__tl_build_put:nw #1#2 \__tl_build_last:NNn #3#4#5
  { #2 \__tl_build_last:NNn #3 #4 { #1 #5 } }
```

(*End of definition for `\tl_build_put_right:Nn` and others. These functions are documented on page 128.*)

`\tl_build_put_left:Nn`
`\tl_build_put_left:Ne`
`\tl_build_put_left:Nx`
`\tl_build_gput_left:Nn`
`\tl_build_gput_left:Ne`
`\tl_build_gput_left:Nx`
`\__tl_build_put_left:NNn`

See `\tl_build_put_right:Nn` for all the machinery. We could easily provide `\tl_build_put_left_right:Nnn`, by just adding the ⟨*right*⟩ material after the `{`⟨*left*⟩`}` in the e-expanding assignment.

```
\cs_new_protected:Npn \tl_build_put_left:Nn #1
  { \__tl_build_put_left:NNn \cs_set_nopar:Npe #1 }
\cs_generate_variant:Nn \tl_build_put_left:Nn { Ne , Nx }
\cs_new_protected:Npn \tl_build_gput_left:Nn #1
  { \__tl_build_put_left:NNn \cs_gset_nopar:Npe #1 }
\cs_generate_variant:Nn \tl_build_gput_left:Nn { Ne , Nx }
\cs_new_protected:Npn \__tl_build_put_left:NNn #1#2#3
  {
    #1 #2
      {
        \__kernel_exp_not:w \exp_after:wN
          {
            \exp:w \exp_after:wN \__tl_build_put:nn
              \exp_after:wN {#2} {#3}
          }
      }
  }
```

(*End of definition for* `\tl_build_put_left:Nn`*,* `\tl_build_gput_left:Nn`*, and* `\__tl_build_put_left:NNn`*. These functions are documented on page 128.*)

`\tl_build_end:N`
`\tl_build_gend:N`
`\__tl_build_end_loop:NN`

Get the data then clear the ⟨*next tl*⟩ recursively until finding an empty one. It is perhaps wasteful to repeatedly use `\cs_to_str:N`. The local/global scope is checked by `\tl_set:Ne` or `\tl_gset:Ne`.

```
\cs_new_protected:Npn \tl_build_end:N #1
  {
    \__tl_build_get:NNN \__kernel_tl_set:Nx #1 #1
    \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } \tl_clear:N
  }
\cs_new_protected:Npn \tl_build_gend:N #1
  {
    \__tl_build_get:NNN \__kernel_tl_gset:Nx #1 #1
    \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } \tl_gclear:N
  }
\cs_new_protected:Npn \__tl_build_end_loop:NN #1#2
  {
    \if_meaning:w \c_empty_tl #1
      \exp_after:wN \use_none:nnnnnn
    \fi:
    #2 #1
    \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } #2
  }
```

(*End of definition for* `\tl_build_end:N`*,* `\tl_build_gend:N`*, and* `\__tl_build_end_loop:NN`*. These functions are documented on page 129.*)

`\tl_build_get_intermediate:NN`

```
\cs_new_protected:Npn \tl_build_get_intermediate:NN
  { \__tl_build_get:NNN \__kernel_tl_set:Nx }
```

(*End of definition for* `\tl_build_get_intermediate:NN`*. This function is documented on page 129.*)

`\__tl_build_get:NNN`
`\__tl_build_get:w`
`\__tl_build_get_end:w`

The idea is to expand the ⟨*tl var*⟩ then the ⟨*next tl*⟩ and so on, all within an e-expanding assignment, and wrap as appropriate in `\exp_not:n`. The various ⟨*left*⟩ parts are left in the assignment as we go, which enables us to expand the ⟨*next tl*⟩ at the right place. The various ⟨*right*⟩ parts are eventually picked up in one last `\exp_not:n`, with a brace trick to wrap all the ⟨*right*⟩ parts together.

```
\cs_new_protected:Npn \__tl_build_get:NNN #1#2#3
  { #1 #3 { \if_false: { \fi: \exp_after:wN \__tl_build_get:w #2 } } }
\cs_new:Npn \__tl_build_get:w #1 \__tl_build_last:NNn #2#3#4
  {
    \exp_not:n {#4}
    \if_meaning:w \c_empty_tl #3
      \exp_after:wN \__tl_build_get_end:w
    \fi:
    \exp_after:wN \__tl_build_get:w #3
  }
\cs_new:Npn \__tl_build_get_end:w #1#2#3
  { \__kernel_exp_not:w \exp_after:wN { \if_false: } \fi: }
```

(*End of definition for* `\__tl_build_get:NNN`*,* `\__tl_build_get:w`*, and* `\__tl_build_get_end:w`*.*)

```
⟨/package⟩
```

Chapter 55

l3str implementation

```
⟨*package⟩
⟨@@=str⟩
```

55.1 Internal auxiliaries

\s__str_mark
\s__str_stop

Internal scan marks.

```
\scan_new:N \s__str_mark
\scan_new:N \s__str_stop
```

(*End of definition for* `\s__str_mark` *and* `\s__str_stop`.)

__str_use_none_delimit_by_s_stop:w
__str_use_i_delimit_by_s_stop:nw

Functions to gobble up to a scan mark.

```
\cs_new:Npn \__str_use_none_delimit_by_s_stop:w #1 \s__str_stop { }
\cs_new:Npn \__str_use_i_delimit_by_s_stop:nw #1 #2 \s__str_stop {#1}
```

(*End of definition for* `\__str_use_none_delimit_by_s_stop:w` *and* `\__str_use_i_delimit_by_s_stop:nw`.)

\q__str_recursion_tail
\q__str_recursion_stop

Internal recursion quarks.

```
\quark_new:N \q__str_recursion_tail
\quark_new:N \q__str_recursion_stop
```

(*End of definition for* `\q__str_recursion_tail` *and* `\q__str_recursion_stop`.)

__str_if_recursion_tail_break:NN
__str_if_recursion_tail_stop_do:Nn

Functions to query recursion quarks.

```
\__kernel_quark_new_test:N \__str_if_recursion_tail_break:NN
\__kernel_quark_new_test:N \__str_if_recursion_tail_stop_do:Nn
```

(*End of definition for* `\__str_if_recursion_tail_break:NN` *and* `\__str_if_recursion_tail_stop_do:Nn`.)

55.2 Creating and setting string variables

\str_new:N
\str_new:c
\str_use:N
\str_use:c
\str_clear:N
\str_clear:c
\str_gclear:N
\str_gclear:c
\str_clear_new:N
\str_clear_new:c
\str_gclear_new:N
\str_gclear_new:c
\str_set_eq:NN
\str_set_eq:cN
\str_set_eq:Nc
\str_set_eq:cc
\str_gset_eq:NN
\str_gset_eq:cN
\str_gset_eq:Nc
\str_gset_eq:cc
\str_concat:NNN
\str_concat:ccc
\str_gconcat:NNN
\str_gconcat:ccc

A string is simply a token list. The full mapping system isn't set up yet so do things by hand.

```
\group_begin:
  \cs_set_protected:Npn \__str_tmp:n #1
    {
      \tl_if_blank:nF {#1}
        {
          \cs_new_eq:cc { str_ #1 :N } { tl_ #1 :N }
          \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :N } { c }
          \__str_tmp:n
        }
    }
  \__str_tmp:n
    { new }
    { use }
    { clear }
    { gclear }
    { clear_new }
    { gclear_new }
    { }
\group_end:
\cs_new_eq:NN \str_set_eq:NN \tl_set_eq:NN
\cs_new_eq:NN \str_gset_eq:NN \tl_gset_eq:NN
\cs_generate_variant:Nn \str_set_eq:NN  { c , Nc , cc }
\cs_generate_variant:Nn \str_gset_eq:NN { c , Nc , cc }
\cs_new_eq:NN \str_concat:NNN \tl_concat:NNN
\cs_new_eq:NN \str_gconcat:NNN \tl_gconcat:NNN
\cs_generate_variant:Nn \str_concat:NNN  { ccc }
\cs_generate_variant:Nn \str_gconcat:NNN { ccc }
```

(*End of definition for* `\str_new:N` *and others. These functions are documented on page* 131.)

\str_set:Nn
\str_set:NV
\str_set:Ne
\str_set:Nx
\str_set:cn
\str_set:cV
\str_set:ce
\str_set:cx
\str_gset:Nn
\str_gset:NV
\str_gset:Ne
\str_gset:Nx
\str_gset:cn
\str_gset:cV
\str_gset:ce
\str_gset:cx
\str_const:Nn
\str_const:NV
\str_const:Ne
\str_const:Nx
\str_const:cn
\str_const:cV
\str_const:ce
\str_const:cx
\str_put_left:Nn
\str_put_left:NV
\str_put_left:Ne
\str_put_left:Nx
\str_put_left:cn

Similar to corresponding l3tl base functions, except that `\__kernel_exp_not:w` is replaced with `\__kernel_tl_to_str:w`. Just like token list, string constants use `\cs_gset_nopar:Npe` instead of `\__kernel_tl_gset:Nx` so that the scope checking for `c` is applied when l3debug is used. To maintain backward compatibility, in `\str_(g)put_left:Nn` and `\str_(g)put_right:Nn`, contents of string variables are wrapped in `\__kernel_exp_not:w` to prevent further expansion.

```
\cs_new_protected:Npn \str_set:Nn #1#2
  { \__kernel_tl_set:Nx #1 { \__kernel_tl_to_str:w {#2} } }
\cs_gset_protected:Npn \str_gset:Nn #1#2
  { \__kernel_tl_gset:Nx #1 { \__kernel_tl_to_str:w {#2} } }
\cs_new_protected:Npn \str_const:Nn #1#2
  {
    \__kernel_chk_if_free_cs:N #1
    \cs_gset_nopar:Npe #1 { \__kernel_tl_to_str:w {#2} }
  }
\cs_new_protected:Npn \str_put_left:Nn #1#2
  {
    \__kernel_tl_set:Nx #1
      { \__kernel_tl_to_str:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
  }
```

```
\cs_new_protected:Npn \str_gput_left:Nn #1#2
  {
    \__kernel_tl_gset:Nx #1
      { \__kernel_tl_to_str:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
  }
\cs_new_protected:Npn \str_put_right:Nn #1#2
  {
    \__kernel_tl_set:Nx #1
      { \__kernel_exp_not:w \exp_after:wN {#1} \__kernel_tl_to_str:w {#2} }
  }
\cs_new_protected:Npn \str_gput_right:Nn #1#2
  {
    \__kernel_tl_gset:Nx #1
      { \__kernel_exp_not:w \exp_after:wN {#1} \__kernel_tl_to_str:w {#2} }
  }
\cs_generate_variant:Nn \str_set:Nn        { NV , Ne , Nx , c , cV , ce , cx }
\cs_generate_variant:Nn \str_gset:Nn       { NV , Ne , Nx , c , cV , ce , cx }
\cs_generate_variant:Nn \str_const:Nn      { NV , Ne , Nx , c , cV , ce , cx }
\cs_generate_variant:Nn \str_put_left:Nn   { NV , Ne , Nx , c , cV , ce , cx }
\cs_generate_variant:Nn \str_gput_left:Nn  { NV , Ne , Nx , c , cV , ce , cx }
\cs_generate_variant:Nn \str_put_right:Nn  { NV , Ne , Nx , c , cV , ce , cx }
\cs_generate_variant:Nn \str_gput_right:Nn { NV , Ne , Nx , c , cV , ce , cx }
```

(*End of definition for* `\str_set:Nn` *and others. These functions are documented on page 132.*)

55.3 Modifying string variables

`\str_replace_all:Nnn`
`\str_replace_all:cnn`
`\str_greplace_all:Nnn`
`\str_greplace_all:cnn`
`\str_replace_once:Nnn`
`\str_replace_once:cnn`
`\str_greplace_once:Nnn`
`\str_greplace_once:cnn`
`\__str_replace:NNNnn`
`\__str_replace_aux:NNNnnn`
`\__str_replace_next:w`

Start by applying `\tl_to_str:n` to convert the old and new token lists to strings, and also apply `\tl_to_str:N` to avoid any issues if we are fed a token list variable. Then the code is a much simplified version of the token list code because neither the delimiter nor the replacement can contain macro parameters or braces. The delimiter `\s__str_mark` cannot appear in the string to edit so it is used in all cases. Some `e`-expansion is unnecessary. There is no need to avoid losing braces nor to protect against expansion. The ending code is much simplified and does not need to hide in braces.

```
\cs_new_protected:Npn \str_replace_once:Nnn
  { \__str_replace:NNNnn \prg_do_nothing: \__kernel_tl_set:Nx  }
\cs_new_protected:Npn \str_greplace_once:Nnn
  { \__str_replace:NNNnn \prg_do_nothing: \__kernel_tl_gset:Nx }
\cs_new_protected:Npn \str_replace_all:Nnn
  { \__str_replace:NNNnn \__str_replace_next:w \__kernel_tl_set:Nx  }
\cs_new_protected:Npn \str_greplace_all:Nnn
  { \__str_replace:NNNnn \__str_replace_next:w \__kernel_tl_gset:Nx }
\cs_generate_variant:Nn \str_replace_once:Nnn  { c }
\cs_generate_variant:Nn \str_greplace_once:Nnn { c }
\cs_generate_variant:Nn \str_replace_all:Nnn   { c }
\cs_generate_variant:Nn \str_greplace_all:Nnn  { c }
\cs_new_protected:Npn \__str_replace:NNNnn #1#2#3#4#5
  {
    \tl_if_empty:nTF {#4}
      {
        \msg_error:nne { kernel } { empty-search-pattern } {#5}
      }
      {
```

```
        \use:e
          {
            \exp_not:n { \__str_replace_aux:NNNnnn #1 #2 #3 }
              { \tl_to_str:N #3 }
              { \tl_to_str:n {#4} } { \tl_to_str:n {#5} }
          }
      }
  }
\cs_new_protected:Npn \__str_replace_aux:NNNnnn #1#2#3#4#5#6
  {
    \cs_set:Npn \__str_replace_next:w ##1 #5 { ##1 #6 #1 }
    #2 #3
      {
        \__str_replace_next:w
        #4
        \__str_use_none_delimit_by_s_stop:w
        #5
        \s__str_stop
      }
  }
\cs_new_eq:NN \__str_replace_next:w ?
```

(*End of definition for* `\str_replace_all:Nnn` *and others. These functions are documented on page 139.*)

`\str_remove_once:Nn`
`\str_remove_once:cn`
`\str_gremove_once:Nn`
`\str_gremove_once:cn`

Removal is just a special case of replacement.

```
\cs_new_protected:Npn \str_remove_once:Nn #1#2
  { \str_replace_once:Nnn #1 {#2} { } }
\cs_new_protected:Npn \str_gremove_once:Nn #1#2
  { \str_greplace_once:Nnn #1 {#2} { } }
\cs_generate_variant:Nn \str_remove_once:Nn  { c }
\cs_generate_variant:Nn \str_gremove_once:Nn { c }
```

(*End of definition for* `\str_remove_once:Nn` *and* `\str_gremove_once:Nn`*. These functions are documented on page 139.*)

`\str_remove_all:Nn`
`\str_remove_all:cn`
`\str_gremove_all:Nn`
`\str_gremove_all:cn`

Removal is just a special case of replacement.

```
\cs_new_protected:Npn \str_remove_all:Nn #1#2
  { \str_replace_all:Nnn #1 {#2} { } }
\cs_new_protected:Npn \str_gremove_all:Nn #1#2
  { \str_greplace_all:Nnn #1 {#2} { } }
\cs_generate_variant:Nn \str_remove_all:Nn  { c }
\cs_generate_variant:Nn \str_gremove_all:Nn { c }
```

(*End of definition for* `\str_remove_all:Nn` *and* `\str_gremove_all:Nn`*. These functions are documented on page 139.*)

55.4 String comparisons

`\str_if_empty_p:N`
`\str_if_empty_p:c`
`\str_if_empty:NTF`
`\str_if_empty:cTF`
`\str_if_empty_p:n`
`\str_if_empty:nTF`
`\str_if_exist_p:N`
`\str_if_exist_p:c`
`\str_if_exist:NTF`
`\str_if_exist:cTF`

More copy-paste!

```
\prg_new_eq_conditional:NNn \str_if_exist:N \tl_if_exist:N
  { p , T , F , TF }
\prg_new_eq_conditional:NNn \str_if_exist:c \tl_if_exist:c
  { p , T , F , TF }
\prg_new_eq_conditional:NNn \str_if_empty:N \tl_if_empty:N
```

```
  { p , T , F , TF }
\prg_new_eq_conditional:NNn \str_if_empty:c \tl_if_empty:c
  { p , T , F , TF }
\prg_new_eq_conditional:NNn \str_if_empty:n \tl_if_empty:n
  { p , T , F , TF }
```

(*End of definition for* `\str_if_empty:NTF`*,* `\str_if_empty:nTF`*, and* `\str_if_exist:NTF`*. These functions are documented on page 132.*)

`\__str_if_eq:nn`

String comparisons rely on the primitive `\(pdf)strcmp`, so we define a new name for it.

```
\cs_new_eq:NN \__str_if_eq:nn \tex_strcmp:D
```

(*End of definition for* `\__str_if_eq:nn`*.*)

`\str_compare_p:nNn`
`\str_compare_p:eNe`
`\str_compare:nNnTF`
`\str_compare:eNeTF`

Simply rely on `\__str_if_eq:nn`, which expands to `-1`, `0` or `1`. The `ee` version is created directly because it is more efficient.

```
\prg_new_conditional:Npnn \str_compare:nNn #1#2#3 { p , T , F , TF }
  {
    \if_int_compare:w
      \__str_if_eq:nn { \exp_not:n {#1} } { \exp_not:n {#3} }
      #2 \c_zero_int
      \prg_return_true: \else: \prg_return_false: \fi:
  }
\prg_new_conditional:Npnn \str_compare:eNe #1#2#3 { p , T , F , TF }
  {
    \if_int_compare:w \__str_if_eq:nn {#1} {#3} #2 \c_zero_int
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End of definition for* `\str_compare:nNnTF`*. This function is documented on page 134.*)

`\str_if_eq_p:nn`
`\str_if_eq_p:Vn`
`\str_if_eq_p:on`
`\str_if_eq_p:nV`
`\str_if_eq_p:no`
`\str_if_eq_p:VV`
`\str_if_eq_p:ee`
`\str_if_eq:nnTF`
`\str_if_eq:VnTF`
`\str_if_eq:onTF`
`\str_if_eq:nVTF`
`\str_if_eq:noTF`
`\str_if_eq:VVTF`
`\str_if_eq:eeTF`

Modern engines provide a direct way of comparing two token lists, but returning a number. This set of conditionals therefore makes life a bit clearer. The `nn` and `ee` versions are created directly as this is most efficient. Since `\__str_if_eq:nn` will expand to `0` as an explicit character with category 12 if the two lists match (and either `-1` or `1` if they don't) we can use `\if:w` here which is faster than using `\if_int_compare:w`.

```
\prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
  {
    \if:w 0 \__str_if_eq:nn { \exp_not:n {#1} } { \exp_not:n {#2} }
      \prg_return_true: \else: \prg_return_false: \fi:
  }
\prg_generate_conditional_variant:Nnn \str_if_eq:nn
  { V , v , o , nV , no , VV , nv } { p , T , F , TF }
\prg_new_conditional:Npnn \str_if_eq:ee #1#2 { p , T , F , TF }
  {
    \if:w 0 \__str_if_eq:nn {#1} {#2}
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End of definition for* `\str_if_eq:nnTF`*. This function is documented on page 133.*)

`\str_if_eq_p:NN`
`\str_if_eq_p:Nc`
`\str_if_eq_p:cN`
`\str_if_eq_p:cc`
`\str_if_eq:NNTF`
`\str_if_eq:NcTF`
`\str_if_eq:cNTF`
`\str_if_eq:ccTF`

Note that `\str_if_eq:NNTF` is different from `\tl_if_eq:NNTF` because it needs to ignore category codes.

```
\prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
  {
```

```
    \if:w 0 \__str_if_eq:nn { \tl_to_str:N #1 } { \tl_to_str:N #2 }
      \prg_return_true: \else: \prg_return_false: \fi:
  }
\prg_generate_conditional_variant:Nnn \str_if_eq:NN
  { c , Nc , cc } { T , F , TF , p }
```

(*End of definition for* `\str_if_eq:NNTF`*. This function is documented on page 132.*)

`\str_if_in:NnTF`
`\str_if_in:cnTF`
`\str_if_in:nnTF`

Everything here needs to be detokenized but beyond that it is a simple token list test. It would be faster to fine-tune the `T`, `F`, `TF` variants by calling the appropriate variant of `\tl_if_in:nnTF` directly but that takes more code.

```
\prg_new_protected_conditional:Npnn \str_if_in:Nn #1#2 { T , F , TF }
  {
    \use:e
      { \tl_if_in:nnTF { \tl_to_str:N #1 } { \tl_to_str:n {#2} } }
      { \prg_return_true: } { \prg_return_false: }
  }
\prg_generate_conditional_variant:Nnn \str_if_in:Nn
  { c } { T , F , TF }
\prg_new_protected_conditional:Npnn \str_if_in:nn #1#2 { T , F , TF }
  {
    \use:e
      { \tl_if_in:nnTF { \tl_to_str:n {#1} } { \tl_to_str:n {#2} } }
      { \prg_return_true: } { \prg_return_false: }
  }
```

(*End of definition for* `\str_if_in:NnTF` *and* `\str_if_in:nnTF`*. These functions are documented on page 133.*)

`\str_case:nn`
`\str_case:Vn`
`\str_case:on`
`\str_case:en`
`\str_case:nV`
`\str_case:nv`
`\str_case:nnTF`
`\str_case:VnTF`
`\str_case:onTF`
`\str_case:enTF`
`\str_case:nVTF`
`\str_case:nvTF`
`\str_case:Nn`
`\str_case:NnTF`
`\str_case_e:nn`
`\str_case_e:en`
`\str_case_e:nnTF`
`\str_case_e:enTF`
`\__str_case:nnTF`
`\__str_case_e:nnTF`
`\__str_case:nw`
`\__str_case_e:nw`
`\__str_case_end:nw`

The aim here is to allow the case statement to be evaluated using a known number of expansion steps (two), and without needing to use an explicit "end of recursion" marker. That is achieved by using the test input as the final case, as this is always true. The trick is then to tidy up the output such that the appropriate case code plus either the `true` or `false` branch code is inserted.

```
\cs_new:Npn \str_case:nn #1#2
  {
    \exp:w
    \__str_case:nnTF {#1} {#2} { } { }
  }
\cs_new:Npn \str_case:nnT #1#2#3
  {
    \exp:w
    \__str_case:nnTF {#1} {#2} {#3} { }
  }
\cs_new:Npn \str_case:nnF #1#2
  {
    \exp:w
    \__str_case:nnTF {#1} {#2} { }
  }
\cs_new:Npn \str_case:nnTF #1#2
  {
    \exp:w
    \__str_case:nnTF {#1} {#2}
  }
```

```
\cs_new:Npn \__str_case:nnTF #1#2#3#4
  { \__str_case:nw {#1} #2 {#1} { } \s__str_mark {#3} \s__str_mark {#4} \s__str_stop }
\cs_generate_variant:Nn \str_case:nn   { V , o , e , nV , nv }
\prg_generate_conditional_variant:Nnn \str_case:nn
  { V , o , e , nV , nv } { T , F , TF }
\cs_new_eq:NN \str_case:Nn   \str_case:Vn
\cs_new_eq:NN \str_case:NnT  \str_case:VnT
\cs_new_eq:NN \str_case:NnF  \str_case:VnF
\cs_new_eq:NN \str_case:NnTF \str_case:VnTF
\cs_new:Npn \__str_case:nw #1#2#3
  {
    \str_if_eq:nnTF {#1} {#2}
      { \__str_case_end:nw {#3} }
      { \__str_case:nw {#1} }
  }
\cs_new:Npn \str_case_e:nn #1#2
  {
    \exp:w
    \__str_case_e:nnTF {#1} {#2} { } { }
  }
\cs_new:Npn \str_case_e:nnT #1#2#3
  {
    \exp:w
    \__str_case_e:nnTF {#1} {#2} {#3} { }
  }
\cs_new:Npn \str_case_e:nnF #1#2
  {
    \exp:w
    \__str_case_e:nnTF {#1} {#2} { }
  }
\cs_new:Npn \str_case_e:nnTF #1#2
  {
    \exp:w
    \__str_case_e:nnTF {#1} {#2}
  }
\cs_new:Npn \__str_case_e:nnTF #1#2#3#4
  { \__str_case_e:nw {#1} #2 {#1} { } \s__str_mark {#3} \s__str_mark {#4} \s__str_stop }
\cs_generate_variant:Nn \str_case_e:nn { e }
\prg_generate_conditional_variant:Nnn \str_case_e:nn { e } { T , F , TF }
\cs_new:Npn \__str_case_e:nw #1#2#3
  {
    \str_if_eq:eeTF {#1} {#2}
      { \__str_case_end:nw {#3} }
      { \__str_case_e:nw {#1} }
  }
```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then `#1` is the code to insert, `#2` is the *next* case to check on and `#3` is all of the rest of the cases code. That means that `#4` is the `true` branch code, and `#5` tidies up the spare `\s__str_mark` and the `false` branch. On the other hand, if none of the cases matched then we arrive here using the "termination" case of comparing the search with itself. That means that `#1` is empty, `#2` is the first `\s__str_mark` and so `#4` is the `false` code (the `true` code is mopped up by `#3`).

```
\cs_new:Npn \__str_case_end:nw #1#2#3 \s__str_mark #4#5 \s__str_stop
```

```
{ \exp_end: #1 #4 }
```

(*End of definition for* `\str_case:nnTF` *and others. These functions are documented on page* *133*.)

55.5 Mapping over strings

`\str_map_function:NN`
`\str_map_function:cN`
`\str_map_function:nN`
`\str_map_inline:Nn`
`\str_map_inline:cn`
`\str_map_inline:nn`
`\str_map_variable:NNn`
`\str_map_variable:cNn`
`\str_map_variable:nNn`
`\str_map_break:`
`\str_map_break:n`
`\__str_map_function:w`
`\__str_map_function:nn`
`\__str_map_inline:NN`
`\__str_map_variable:NnN`

The inline and variable mappings are similar to the usual token list mappings but start out by turning the argument to an "other string". Doing the same for the expandable function mapping would require `\__kernel_str_to_other:n`, quadratic in the string length. To deal with spaces in that case, `\__str_map_function:w` replaces the following space by a braced space and a further call to itself. These are received by `\__str_map_function:nn`, which passes the space to `#1` and calls `\__str_map_function:w` to deal with the next space. The space before the braced space allows to optimize the `\q__str_recursion_tail` test. Of course we need to include a trailing space (the question mark is needed to avoid losing the space when TeX tokenizes the line). At the cost of about three more auxiliaries this code could get a 9 times speed up by testing only every 9-th character for whether it is `\q__str_recursion_tail` (also by converting 9 spaces at a time in the `\str_map_function:nN` case).

For the `map_variable` functions we use a string assignment to store each character because spaces are made catcode 12 before the loop.

```
\cs_new:Npn \str_map_function:nN #1#2
  {
    \exp_after:wN \__str_map_function:w
    \exp_after:wN \__str_map_function:nn \exp_after:wN #2
      \__kernel_tl_to_str:w {#1}
      \q__str_recursion_tail ? ~
    \prg_break_point:Nn \str_map_break: { }
  }
\cs_new:Npn \str_map_function:NN
  { \exp_args:No \str_map_function:nN }
\cs_new:Npn \__str_map_function:w #1 ~
  { #1 { ~ { ~ } \__str_map_function:w } }
\cs_new:Npn \__str_map_function:nn #1#2
  {
    \if_meaning:w \q__str_recursion_tail #2
      \exp_after:wN \str_map_break:
    \fi:
    #1 #2 \__str_map_function:nn {#1}
  }
\cs_generate_variant:Nn \str_map_function:NN { c }
\cs_new_protected:Npn \str_map_inline:nn #1#2
  {
    \int_gincr:N \g__kernel_prg_map_int
    \cs_gset_protected:cpn
      { __str_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
    \use:e
      {
        \exp_not:N \__str_map_inline:NN
        \exp_not:c { __str_map_ \int_use:N \g__kernel_prg_map_int :w }
        \__kernel_str_to_other_fast:n {#1}
      }
      \q__str_recursion_tail
```

```
    \prg_break_point:Nn \str_map_break:
      { \int_gdecr:N \g__kernel_prg_map_int }
  }
\cs_new_protected:Npn \str_map_inline:Nn
  { \exp_args:No \str_map_inline:nn }
\cs_generate_variant:Nn \str_map_inline:Nn { c }
\cs_new:Npn \__str_map_inline:NN #1#2
  {
    \__str_if_recursion_tail_break:NN #2 \str_map_break:
    \exp_args:No #1 { \token_to_str:N #2 }
    \__str_map_inline:NN #1
  }
\cs_new_protected:Npn \str_map_variable:nNn #1#2#3
  {
    \use:e
      {
        \exp_not:n { \__str_map_variable:NnN #2 {#3} }
        \__kernel_str_to_other_fast:n {#1}
      }
      \q__str_recursion_tail
    \prg_break_point:Nn \str_map_break: { }
  }
\cs_new_protected:Npn \str_map_variable:NNn
  { \exp_args:No \str_map_variable:nNn }
\cs_new_protected:Npn \__str_map_variable:NnN #1#2#3
  {
    \__str_if_recursion_tail_break:NN #3 \str_map_break:
    \str_set:Nn #1 {#3}
    \use:n {#2}
    \__str_map_variable:NnN #1 {#2}
  }
\cs_generate_variant:Nn \str_map_variable:NNn { c }
\cs_new:Npn \str_map_break:
  { \prg_map_break:Nn \str_map_break: { } }
\cs_new:Npn \str_map_break:n
  { \prg_map_break:Nn \str_map_break: }
```

(*End of definition for* `\str_map_function:NN` *and others. These functions are documented on page* 134.)

`\str_map_tokens:Nn`
`\str_map_tokens:cn`
`\str_map_tokens:nn`

Uses an auxiliary of `\str_map_function:NN`.

```
\cs_new:Npn \str_map_tokens:nn #1#2
  {
    \exp_args:Nno \use:nn
      { \__str_map_function:w \__str_map_function:nn {#2} }
      { \__kernel_tl_to_str:w {#1} }
      \q__str_recursion_tail ? ~
    \prg_break_point:Nn \str_map_break: { }
  }
\cs_new:Npn \str_map_tokens:Nn { \exp_args:No \str_map_tokens:nn }
\cs_generate_variant:Nn \str_map_tokens:Nn { c }
```

(*End of definition for* `\str_map_tokens:Nn` *and* `\str_map_tokens:nn`. *These functions are documented on page* 135.)

55.6 Accessing specific characters in a string

__kernel_str_to_other:n
__str_to_other_loop:w
__str_to_other_end:w

First apply `\tl_to_str:n`, then replace all spaces by "other" spaces, 8 at a time, storing the converted part of the string between the `\s__str_mark` and `\s__str_stop` markers. The end is detected when `\__str_to_other_loop:w` finds one of the trailing `A`, distinguished from any contents of the initial token list by their category. Then `\__str_to_other_end:w` is called, and finds the result between `\s__str_mark` and the first `A` (well, there is also the need to remove a space).

```
\cs_new:Npn \__kernel_str_to_other:n #1
  {
    \exp_after:wN \__str_to_other_loop:w
      \tl_to_str:n {#1} ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \s__str_mark \s__str_stop
  }
\group_begin:
\tex_lccode:D ‘\* = ‘\  %
\tex_lccode:D ‘\A = ‘\A %
\tex_lowercase:D
  {
    \group_end:
    \cs_new:Npn \__str_to_other_loop:w
      #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \s__str_stop
      {
        \if_meaning:w A #8
          \__str_to_other_end:w
        \fi:
        \__str_to_other_loop:w
        #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \s__str_stop
      }
    \cs_new:Npn \__str_to_other_end:w \fi: #1 \s__str_mark #2 * A #3 \s__str_stop
      { \fi: #2 }
  }
```

(*End of definition for* `\__kernel_str_to_other:n`*,* `\__str_to_other_loop:w`*, and* `\__str_to_other_end:w`*.*)

__kernel_str_to_other_fast:n
__kernel_str_to_other_fast_loop:w
__str_to_other_fast_end:w

The difference with `\__kernel_str_to_other:n` is that the converted part is left in the input stream, making these commands only restricted-expandable.

```
\cs_new:Npn \__kernel_str_to_other_fast:n #1
  {
    \exp_after:wN \__str_to_other_fast_loop:w \tl_to_str:n {#1} ~
      A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \s__str_stop
  }
\group_begin:
\tex_lccode:D ‘\* = ‘\  %
\tex_lccode:D ‘\A = ‘\A %
\tex_lowercase:D
  {
    \group_end:
    \cs_new:Npn \__str_to_other_fast_loop:w
      #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 ~
      {
        \if_meaning:w A #9
          \__str_to_other_fast_end:w
        \fi:
```

```
      #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * #9
      \__str_to_other_fast_loop:w *
    }
  \cs_new:Npn \__str_to_other_fast_end:w #1 * A #2 \s__str_stop {#1}
}
```

(*End of definition for* `\__kernel_str_to_other_fast:n` *,* `\__kernel_str_to_other_fast_loop:w` *, and* `\__str_to_other_fast_end:w` *.*)

`\str_item:Nn` `\str_item:cn` `\str_item:nn` `\str_item_ignore_spaces:nn` `\__str_item:nn` `\__str_item:w`

The `\str_item:nn` hands its argument with spaces escaped to `\__str_item:nn`, and makes sure to turn the result back into a proper string (with category code 10 spaces) eventually. The `\str_item_ignore_spaces:nn` function does not escape spaces, which are thus ignored by `\__str_item:nn` since everything else is done with undelimited arguments. Evaluate the ⟨*index*⟩ argument `#2` and count characters in the string, passing those two numbers to `\__str_item:w` for further analysis. If the ⟨*index*⟩ is negative, shift it by the ⟨*count*⟩ to know the how many character to discard, and if that is still negative give an empty result. If the ⟨*index*⟩ is larger than the ⟨*count*⟩, give an empty result, and otherwise discard ⟨*index*⟩ $-$ 1 characters before returning the following one. The shift by -1 is obtained by inserting an empty brace group before the string in that case: that brace group also covers the case where the ⟨*index*⟩ is zero.

```
\cs_new:Npn \str_item:Nn { \exp_args:No \str_item:nn }
\cs_generate_variant:Nn \str_item:Nn { c }
\cs_new:Npn \str_item:nn #1#2
  {
    \exp_args:Nf \tl_to_str:n
      {
        \exp_args:Nf \__str_item:nn
          { \__kernel_str_to_other:n {#1} } {#2}
      }
  }
\cs_new:Npn \str_item_ignore_spaces:nn #1
  { \exp_args:No \__str_item:nn { \tl_to_str:n {#1} } }
\cs_new:Npn \__str_item:nn #1#2
  {
    \exp_after:wN \__str_item:w
    \int_value:w \int_eval:n {#2} \exp_after:wN ;
    \int_value:w \__str_count:n {#1} ;
    #1 \s__str_stop
  }
\cs_new:Npn \__str_item:w #1; #2;
  {
    \int_compare:nNnTF {#1} < 0
      {
        \int_compare:nNnTF {#1} < {-#2}
          { \__str_use_none_delimit_by_s_stop:w }
          {
            \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
            \exp:w \exp_after:wN \__str_skip_exp_end:w
              \int_value:w \int_eval:n { #1 + #2 } ;
          }
      }
      {
        \int_compare:nNnTF {#1} > {#2}
          { \__str_use_none_delimit_by_s_stop:w }
```

```
        {
          \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
          \exp:w \__str_skip_exp_end:w #1 ; { }
        }
    }
}
```

(*End of definition for* `\str_item:Nn` *and others. These functions are documented on page 137.*)

`\__str_skip_exp_end:w`
`\__str_skip_loop:wNNNNNNNN`
`\__str_skip_end:w`
`\__str_skip_end:NNNNNNNN`

Removes `max(#1,0)` characters from the input stream, and then leaves `\exp_end:`. This should be expanded using `\exp:w`. We remove characters 8 at a time until there are at most 8 to remove. Then we do a dirty trick: the `\if_case:w` construction leaves between 0 and 8 times the `\or:` control sequence, and those `\or:` become arguments of `\__str_skip_end:NNNNNNNN`. If the number of characters to remove is 6, say, then there are two `\or:` left, and the 8 arguments of `\__str_skip_end:NNNNNNNN` are the two `\or:`, and 6 characters from the input stream, exactly what we wanted to remove. Then close the `\if_case:w` conditional with `\fi:`, and stop the initial expansion with `\exp_end:` (see places where `\__str_skip_exp_end:w` is called).

```
\cs_new:Npn \__str_skip_exp_end:w #1;
  {
    \if_int_compare:w #1 > 8 \exp_stop_f:
      \exp_after:wN \__str_skip_loop:wNNNNNNNN
    \else:
      \exp_after:wN \__str_skip_end:w
      \int_value:w \int_eval:w
    \fi:
    #1 ;
  }
\cs_new:Npn \__str_skip_loop:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
  {
    \exp_after:wN \__str_skip_exp_end:w
      \int_value:w \int_eval:n { #1 - 8 } ;
  }
\cs_new:Npn \__str_skip_end:w #1 ;
  {
    \exp_after:wN \__str_skip_end:NNNNNNNN
    \if_case:w #1 \exp_stop_f: \or: \or: \or: \or: \or: \or: \or: \or:
  }
\cs_new:Npn \__str_skip_end:NNNNNNNN #1#2#3#4#5#6#7#8 { \fi: \exp_end: }
```

(*End of definition for* `\__str_skip_exp_end:w` *and others.*)

`\str_range:Nnn`
`\str_range:nnn`
`\str_range_ignore_spaces:nnn`
`\__str_range:nnn`
`\__str_range:w`
`\__str_range:nnw`

Sanitize the string. Then evaluate the arguments. At this stage we also decrement the ⟨*start index*⟩, since our goal is to know how many characters should be removed. Then limit the range to be non-negative and at most the length of the string (this avoids needing to check for the end of the string when grabbing characters), shifting negative numbers by the appropriate amount. Afterwards, skip characters, then keep some more, and finally drop the end of the string.

```
\cs_new:Npn \str_range:Nnn { \exp_args:No \str_range:nnn }
\cs_generate_variant:Nn \str_range:Nnn { c }
\cs_new:Npn \str_range:nnn #1#2#3
  {
    \exp_args:Nf \tl_to_str:n
```

```
      {
        \exp_args:Nf \__str_range:nnn
          { \__kernel_str_to_other:n {#1} } {#2} {#3}
      }
  }
\cs_new:Npn \str_range_ignore_spaces:nnn #1
  { \exp_args:No \__str_range:nnn { \tl_to_str:n {#1} } }
\cs_new:Npn \__str_range:nnn #1#2#3
  {
    \exp_after:wN \__str_range:w
    \int_value:w \__str_count:n {#1} \exp_after:wN ;
    \int_value:w \int_eval:n { (#2) - 1 } \exp_after:wN ;
    \int_value:w \int_eval:n {#3} ;
    #1 \s__str_stop
  }
\cs_new:Npn \__str_range:w #1; #2; #3;
  {
    \exp_args:Nf \__str_range:nnw
      { \__str_range_normalize:nn {#2} {#1} }
      { \__str_range_normalize:nn {#3} {#1} }
  }
\cs_new:Npn \__str_range:nnw #1#2
  {
    \exp_after:wN \__str_collect_delimit_by_q_stop:w
    \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ;
    \exp:w \__str_skip_exp_end:w #1 ;
  }
```

(*End of definition for* `\str_range:Nnn` *and others. These functions are documented on page* *138*.)

`\__str_range_normalize:nn` This function converts an ⟨*index*⟩ argument into an explicit position in the string (a result of 0 denoting "out of bounds"). Expects two explicit integer arguments: the ⟨*index*⟩ `#1` and the string count `#2`. If `#1` is negative, replace it by `#1` + `#2` + 1, then limit to the range [0, `#2`].

```
\cs_new:Npn \__str_range_normalize:nn #1#2
  {
    \int_eval:n
      {
        \if_int_compare:w #1 < \c_zero_int
          \if_int_compare:w #1 < -#2 \exp_stop_f:
            0
          \else:
            #1 + #2 + 1
          \fi:
        \else:
          \if_int_compare:w #1 < #2 \exp_stop_f:
            #1
          \else:
            #2
          \fi:
        \fi:
      }
  }
```

(*End of definition for* `\__str_range_normalize:nn`.)

__str_collect_delimit_by_q_stop:w
__str_collect_loop:wn
__str_collect_loop:wnNNNNNNN
__str_collect_end:wn
__str_collect_end:nnnnnnnnw

Collects `max(#1,0)` characters, and removes everything else until `\s__str_stop`. This is somewhat similar to `\__str_skip_exp_end:w`, but accepts integer expression arguments. This time we can only grab 7 characters at a time. At the end, we use an `\if_case:w` trick again, so that the 8 first arguments of `\__str_collect_end:nnnnnnnnw` are some `\or:`, followed by an `\fi:`, followed by `#1` characters from the input stream. Simply leaving this in the input stream closes the conditional properly and the `\or:` disappear.

```
\cs_new:Npn \__str_collect_delimit_by_q_stop:w #1;
  { \__str_collect_loop:wn #1 ; { } }
\cs_new:Npn \__str_collect_loop:wn #1 ;
  {
    \if_int_compare:w #1 > 7 \exp_stop_f:
      \exp_after:wN \__str_collect_loop:wnNNNNNNN
    \else:
      \exp_after:wN \__str_collect_end:wn
    \fi:
    #1 ;
  }
\cs_new:Npn \__str_collect_loop:wnNNNNNNN #1; #2 #3#4#5#6#7#8#9
  {
    \exp_after:wN \__str_collect_loop:wn
    \int_value:w \int_eval:n { #1 - 7 } ;
    { #2 #3#4#5#6#7#8#9 }
  }
\cs_new:Npn \__str_collect_end:wn #1 ;
  {
    \exp_after:wN \__str_collect_end:nnnnnnnnw
    \if_case:w \if_int_compare:w #1 > \c_zero_int
      #1 \else: 0 \fi: \exp_stop_f:
      \or: \or: \or: \or: \or: \or: \fi:
  }
\cs_new:Npn \__str_collect_end:nnnnnnnnw #1#2#3#4#5#6#7#8 #9 \s__str_stop
  { #1#2#3#4#5#6#7#8 }
```

(*End of definition for* `\__str_collect_delimit_by_q_stop:w` *and others.*)

55.7 Counting characters

\str_count_spaces:N
\str_count_spaces:c
\str_count_spaces:n
__str_count_spaces_loop:w

To speed up this function, we grab and discard 9 space-delimited arguments in each iteration of the loop. The loop stops when the last argument is one of the trailing X⟨*number*⟩, and that ⟨*number*⟩ is added to the sum of 9 that precedes, to adjust the result.

```
\cs_new:Npn \str_count_spaces:N
  { \exp_args:No \str_count_spaces:n }
\cs_generate_variant:Nn \str_count_spaces:N { c }
\cs_new:Npn \str_count_spaces:n #1
  {
    \int_eval:n
      {
        \exp_after:wN \__str_count_spaces_loop:w
        \tl_to_str:n {#1} ~
        X 7 ~ X 6 ~ X 5 ~ X 4 ~ X 3 ~ X 2 ~ X 1 ~ X 0 ~ X -1 ~
        \s__str_stop
```

```
    }
  }
\cs_new:Npn \__str_count_spaces_loop:w #1~#2~#3~#4~#5~#6~#7~#8~#9~
  {
    \if_meaning:w X #9
      \__str_use_i_delimit_by_s_stop:nw
    \fi:
    9 + \__str_count_spaces_loop:w
  }
```

(*End of definition for* `\str_count_spaces:N`*,* `\str_count_spaces:n`*, and* `\__str_count_spaces_loop:w`*. These functions are documented on page 136.*)

`\str_count:N`
`\str_count:c`
`\str_count:n`
`\str_count_ignore_spaces:n`
`\__str_count:n`
`\__str_count_aux:n`
`\__str_count_loop:NNNNNNNNN`

To count characters in a string we could first escape all spaces using `\__kernel_str_to_other:n`, then pass the result to `\tl_count:n`. However, the escaping step would be quadratic in the number of characters in the string, and we can do better. Namely, sum the number of spaces (`\str_count_spaces:n`) and the result of `\tl_count:n`, which ignores spaces. Since strings tend to be longer than token lists, we use specialized functions to count characters ignoring spaces. Namely, loop, grabbing 9 non-space characters at each step, and end as soon as we reach one of the 9 trailing items. The internal function `\__str_count:n`, used in `\str_item:nn` and `\str_range:nnn`, is similar to `\str_count_ignore_spaces:n` but expects its argument to already be a string or a string with spaces escaped.

```
\cs_new:Npn \str_count:N { \exp_args:No \str_count:n }
\cs_generate_variant:Nn \str_count:N { c }
\cs_new:Npn \str_count:n #1
  {
    \__str_count_aux:n
      {
        \str_count_spaces:n {#1}
        + \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1}
      }
  }
\cs_new:Npn \__str_count:n #1
  {
    \__str_count_aux:n
      { \__str_count_loop:NNNNNNNNN #1 }
  }
\cs_new:Npn \str_count_ignore_spaces:n #1
  {
    \__str_count_aux:n
      { \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1} }
  }
\cs_new:Npn \__str_count_aux:n #1
  {
    \int_eval:n
      {
        #1
        { X 8 } { X 7 } { X 6 }
        { X 5 } { X 4 } { X 3 }
        { X 2 } { X 1 } { X 0 }
        \s__str_stop
      }
  }
```

```
\cs_new:Npn \__str_count_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {
    \if_meaning:w X #9
      \exp_after:wN \__str_use_none_delimit_by_s_stop:w
    \fi:
    9 + \__str_count_loop:NNNNNNNNN
  }
```

(*End of definition for* `\str_count:N` *and others. These functions are documented on page 136.*)

55.8 The first character in a string

`\str_head:N`
`\str_head:c`
`\str_head:n`
`\str_head_ignore_spaces:n`
`\__str_head:w`

The `_ignore_spaces` variant applies `\tl_to_str:n` then grabs the first item, thus skipping spaces. As usual, `\str_head:N` expands its argument and hands it to `\str_head:n`. To circumvent the fact that TEX skips spaces when grabbing undelimited macro parameters, `\__str_head:w` takes an argument delimited by a space. If `#1` starts with a non-space character, `\__str_use_i_delimit_by_s_stop:nw` leaves that in the input stream. On the other hand, if `#1` starts with a space, the `\__str_head:w` takes an empty argument, and the single (initially braced) space in the definition of `\__str_head:w` makes its way to the output. Finally, for an empty argument, the (braced) empty brace group in the definition of `\str_head:n` gives an empty result after passing through `\__str_use_i_delimit_by_s_stop:nw`.

```
\cs_new:Npn \str_head:N { \exp_args:No \str_head:n }
\cs_generate_variant:Nn \str_head:N { c }
\cs_new:Npn \str_head:n #1
  {
    \exp_after:wN \__str_head:w
    \tl_to_str:n {#1}
    { { } } ~ \s__str_stop
  }
\cs_new:Npn \__str_head:w #1 ~ %
  { \__str_use_i_delimit_by_s_stop:nw #1 { ~ } }
\cs_new:Npn \str_head_ignore_spaces:n #1
  {
    \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
    \tl_to_str:n {#1} { } \s__str_stop
  }
```

(*End of definition for* `\str_head:N` *and others. These functions are documented on page 137.*)

`\str_tail:N`
`\str_tail:c`
`\str_tail:n`
`\str_tail_ignore_spaces:n`
`\__str_tail_auxi:w`
`\__str_tail_auxii:w`

Getting the tail is a little bit more convoluted than the head of a string. We hit the front of the string with `\reverse_if:N \if_charcode:w \scan_stop:`. This removes the first character, and necessarily makes the test true, since the character cannot match `\scan_stop:`. The auxiliary function then inserts the required `\fi:` to close the conditional, and leaves the tail of the string in the input stream. The details are such that an empty string has an empty tail (this requires in particular that the end-marker `X` be unexpandable and not a control sequence). The `_ignore_spaces` is rather simpler: after converting the input to a string, `\__str_tail_auxii:w` removes one undelimited argument and leaves everything else until an end-marker `\s__str_mark`. One can check that an empty (or blank) string yields an empty tail.

```
\cs_new:Npn \str_tail:N { \exp_args:No \str_tail:n }
\cs_generate_variant:Nn \str_tail:N { c }
```

```
\cs_new:Npn \str_tail:n #1
  {
    \exp_after:wN \__str_tail_auxi:w
    \reverse_if:N \if_charcode:w
        \scan_stop: \tl_to_str:n {#1} X X \s__str_stop
  }
\cs_new:Npn \__str_tail_auxi:w #1 X #2 \s__str_stop { \fi: #1 }
\cs_new:Npn \str_tail_ignore_spaces:n #1
  {
    \exp_after:wN \__str_tail_auxii:w
    \tl_to_str:n {#1} \s__str_mark \s__str_mark \s__str_stop
  }
\cs_new:Npn \__str_tail_auxii:w #1 #2 \s__str_mark #3 \s__str_stop { #2 }
```

(*End of definition for* `\str_tail:N` *and others. These functions are documented on page 137.*)

55.9 String manipulation

`\str_casefold:n`
`\str_casefold:V`
`\str_lowercase:n`
`\str_lowercase:f`
`\str_uppercase:n`
`\str_uppercase:f`
`\__str_change_case:nn`
`\__str_change_case_aux:nn`
`\__str_change_case_result:n`
`\__str_change_case_output:nw`
`\__str_change_case_output:fw`
`\__str_change_case_end:nw`
`\__str_change_case_loop:nw`
`\__str_change_case_space:n`
`\__str_change_case_char:nN`
`\__str_change_case_char_auxi:nN`
`\__str_change_case_char_auxii:nN`
`\__str_change_case_codepoint:nN`
`\__str_change_case_codepoint:nNN`
`\__str_change_case_codepoint:nNNN`
`\__str_change_case_codepoint:nNNNN`
`\__str_change_case_char:nnn`
`\__str_change_case_char_aux:nnn`
`\__str_change_case_char:nnnnn`

Case changing for programmatic reasons is done by first detokenizing input then doing a simple loop that only has to worry about spaces and everything else. The output is detokenized to allow data sharing with text-based case changing. Similarly, for 8-bit engines the multi-byte information is shared.

```
\cs_new:Npn \str_casefold:n  #1 { \__str_change_case:nn {#1} { casefold } }
\cs_new:Npn \str_lowercase:n #1 { \__str_change_case:nn {#1} { lowercase } }
\cs_new:Npn \str_uppercase:n #1 { \__str_change_case:nn {#1} { uppercase } }
\cs_generate_variant:Nn \str_casefold:n  { V }
\cs_generate_variant:Nn \str_lowercase:n { f }
\cs_generate_variant:Nn \str_uppercase:n { f }
\cs_new:Npn \__str_change_case:nn #1
  {
    \exp_after:wN \__str_change_case_aux:nn \exp_after:wN
      { \tl_to_str:n {#1} }
  }
\cs_new:Npn \__str_change_case_aux:nn #1#2
  {
    \__str_change_case_loop:nw {#2} #1 \q__str_recursion_tail \q__str_recursion_stop
      \__str_change_case_result:n { }
  }
\cs_new:Npn \__str_change_case_output:nw #1#2 \__str_change_case_result:n #3
  { #2 \__str_change_case_result:n { #3 #1 } }
\cs_generate_variant:Nn  \__str_change_case_output:nw { f }
\cs_new:Npn \__str_change_case_end:wn #1 \__str_change_case_result:n #2
  { \tl_to_str:n {#2} }
\cs_new:Npn \__str_change_case_loop:nw #1#2 \q__str_recursion_stop
  {
    \tl_if_head_is_space:nTF {#2}
      { \__str_change_case_space:n }
      { \__str_change_case_char:nN }
    {#1} #2 \q__str_recursion_stop
  }
\exp_last_unbraced:NNNNo
  \cs_new:Npn \__str_change_case_space:n #1 \c_space_tl
  {
```

```
    \__str_change_case_output:nw { ~ }
    \__str_change_case_loop:nw {#1}
  }
\cs_new:Npn \__str_change_case_char:nN #1#2
  {
    \__str_if_recursion_tail_stop_do:Nn #2
      { \__str_change_case_end:wn }
    \__str_change_case_codepoint:nN {#1} #2
  }
\if_int_compare:w 0
  \cs_if_exist:NT \tex_XeTeXversion:D { 1 }
  \cs_if_exist:NT \tex_luatexversion:D { 1 }
  > 0 \exp_stop_f:
  \cs_new:Npn \__str_change_case_codepoint:nN #1#2
    { \__str_change_case_char:fnn { \int_eval:n {`#2} } {#1} {#2} }
\else:
    \cs_new:Npe \__str_change_case_codepoint:nN #1#2
      {
        \exp_not:N \int_compare:nNnTF {`#2} > { "80 }
          {
            \cs_if_exist:NTF \tex_pdftexversion:D
              { \exp_not:N \__str_change_case_char_auxi:nN }
              {
                \exp_not:N \int_compare:nNnTF {`#2} > { "FF }
                  { \exp_not:N \__str_change_case_char_auxii:nN }
                  { \exp_not:N \__str_change_case_char_auxi:nN }
              }
          }
          { \exp_not:N \__str_change_case_char_auxii:nN }
            {#1} #2
      }
    \cs_new:Npn \__str_change_case_char_auxi:nN #1#2
      {
        \int_compare:nNnTF {`#2} < { "E0 }
          { \__str_change_case_codepoint:nNN }
          {
            \int_compare:nNnTF {`#2} < { "F0 }
              { \__str_change_case_codepoint:nNNN }
              { \__str_change_case_codepoint:nNNNNN }
          }
            {#1} #2
      }
    \cs_new:Npn \__str_change_case_char_auxii:nN #1#2
      { \__str_change_case_char:fnn { \int_eval:n {`#2} } {#1} {#2} }
    \cs_new:Npn \__str_change_case_codepoint:nNN #1#2#3
      {
        \__str_change_case_char:fnn
          { \int_eval:n { (`#2 - "C0) * "40 + `#3 - "80 } }
          {#1} {#2#3}
      }
    \cs_new:Npn \__str_change_case_codepoint:nNNN #1#2#3#4
      {
        \__str_change_case_char:fnn
          {
```

```
            \int_eval:n
              { (`#2 - "E0) * "1000 + (`#3 - "80) * "40 + `#4 - "80 }
          }
          {#1} {#2#3#4}
      }
    \cs_new:Npn \__str_change_case_codepoint:nNNNN #1#2#3#4#5
      {
        \__str_change_case_char:fnn
          {
            \int_eval:n
              {
                  (`#2 - "F0) * "40000
                + (`#3 - "80) * "1000
                + (`#4 - "80) * "40
                + `#5 - "80
              }
          }
          {#1} {#2#3#4#5}
      }
\fi:
\cs_new:Npn \__str_change_case_char:nnn #1#2#3
  {
    \__str_change_case_output:fw
      {
        \exp_args:Ne \__str_change_case_char_aux:nnn
          { \__kernel_codepoint_case:nn {#2} {#1} } {#1} {#3}
      }
    \__str_change_case_loop:nw {#2}
  }
\cs_generate_variant:Nn \__str_change_case_char:nnn { f }
\cs_new:Npn \__str_change_case_char_aux:nnn #1#2#3
  {
    \use:e { \__str_change_case_char:nnnnn #1 {#2} {#3} }
  }
\cs_new:Npn \__str_change_case_char:nnnnn #1#2#3#4#5
  {
    \int_compare:nNnTF {#1} = {#4}
      { \tl_to_str:n {#5} }
      {
        \codepoint_str_generate:n {#1}
        \tl_if_blank:nF {#2}
          {
            \codepoint_str_generate:n {#2}
            \tl_if_blank:nF {#3}
              { \codepoint_str_generate:n {#3} }
          }
      }
  }
```

(*End of definition for* `\str_casefold:n` *and others. These functions are documented on page 141.*)

`\str_mdfive_hash:n`
`\str_mdfive_hash:e`

```
\cs_new:Npn \str_mdfive_hash:n #1 { \tex_mdfivesum:D { \tl_to_str:n {#1} } }
\cs_new:Npn \str_mdfive_hash:e #1 { \tex_mdfivesum:D {#1} }
```

(*End of definition for* `\str_mdfive_hash:n`*. This function is documented on page 141.*)

\c_ampersand_str
\c_atsign_str
\c_backslash_str
\c_left_brace_str
\c_right_brace_str
\c_circumflex_str
\c_colon_str
\c_dollar_str
\c_hash_str
\c_percent_str
\c_tilde_str
\c_underscore_str
\c_zero_str

For all of those strings, use `\cs_to_str:N` to get characters with the correct category code without worries

```
\str_const:Ne \c_ampersand_str   { \cs_to_str:N \& }
\str_const:Ne \c_atsign_str      { \cs_to_str:N \@ }
\str_const:Ne \c_backslash_str   { \cs_to_str:N \\ }
\str_const:Ne \c_left_brace_str  { \cs_to_str:N \{ }
\str_const:Ne \c_right_brace_str { \cs_to_str:N \} }
\str_const:Ne \c_circumflex_str  { \cs_to_str:N \^ }
\str_const:Ne \c_colon_str       { \cs_to_str:N \: }
\str_const:Ne \c_dollar_str      { \cs_to_str:N \$ }
\str_const:Ne \c_hash_str        { \cs_to_str:N \# }
\str_const:Ne \c_percent_str     { \cs_to_str:N \% }
\str_const:Ne \c_tilde_str       { \cs_to_str:N \~ }
\str_const:Ne \c_underscore_str  { \cs_to_str:N \_ }
\str_const:Ne \c_zero_str        { 0 }
```

(*End of definition for* `\c_ampersand_str` *and others. These variables are documented on page 142.*)

\c_empty_str

An empty string is simply an empty token list.

```
\cs_new_eq:NN \c_empty_str \c_empty_tl
```

(*End of definition for* `\c_empty_str`*. This variable is documented on page 142.*)

\l_tmpa_str
\l_tmpb_str
\g_tmpa_str
\g_tmpb_str

Scratch strings.

```
\str_new:N \l_tmpa_str
\str_new:N \l_tmpb_str
\str_new:N \g_tmpa_str
\str_new:N \g_tmpb_str
```

(*End of definition for* `\l_tmpa_str` *and others. These variables are documented on page 142.*)

55.10 Viewing strings

\str_show:n
\str_show:N
\str_show:c
\str_log:n
\str_log:N
\str_log:c

Displays a string on the terminal.

```
\cs_new_eq:NN \str_show:n \tl_show:n
\cs_new_protected:Npn \str_show:N #1
  {
    \__kernel_chk_tl_type:NnnT #1 { str } { \tl_to_str:N #1 }
      { \tl_show:N #1 }
  }
\cs_generate_variant:Nn \str_show:N { c }
\cs_new_eq:NN \str_log:n \tl_log:n
\cs_new_protected:Npn \str_log:N #1
  {
    \__kernel_chk_tl_type:NnnT #1 { str } { \tl_to_str:N #1 }
      { \tl_log:N #1 }
  }
\cs_generate_variant:Nn \str_log:N { c }
```

(*End of definition for* `\str_show:n` *and others. These functions are documented on page 141.*)

⟨/package⟩

Chapter 56

l3str-convert implementation

```
⟨*package⟩
⟨@@=str⟩
```

56.1 Helpers

56.1.1 Variables and constants

__str_tmp:w
\l__str_internal_tl

Internal scratch space for some functions.

```
\cs_new_protected:Npn \__str_tmp:w { }
\tl_new:N \l__str_internal_tl
```

(*End of definition for* __str_tmp:w *and* \l__str_internal_tl.)

\g__str_result_tl

The \g__str_result_tl variable is used to hold the result of various internal string operations (mostly conversions) which are typically performed in a group. The variable is global so that it remains defined outside the group, to be assigned to a user-provided variable.

```
\tl_new:N \g__str_result_tl
```

(*End of definition for* \g__str_result_tl.)

\c__str_replacement_char_int

When converting, invalid bytes are replaced by the Unicode replacement character "FFFD.

```
\int_const:Nn \c__str_replacement_char_int { "FFFD }
```

(*End of definition for* \c__str_replacement_char_int.)

\c__str_max_byte_int

The maximal byte number.

```
\int_const:Nn \c__str_max_byte_int { 255 }
```

(*End of definition for* \c__str_max_byte_int.)

\s__str

Internal scan marks.

```
\scan_new:N \s__str
```

(*End of definition for* \s__str.)

\q__str_nil Internal quarks.

```
\quark_new:N \q__str_nil
```

(*End of definition for* `\q__str_nil`.)

\g__str_alias_prop To avoid needing one file per encoding/escaping alias, we keep track of those in a property list.

```
\prop_new:N \g__str_alias_prop
\prop_gput:Nnn \g__str_alias_prop { latin1 } { iso88591 }
\prop_gput:Nnn \g__str_alias_prop { latin2 } { iso88592 }
\prop_gput:Nnn \g__str_alias_prop { latin3 } { iso88593 }
\prop_gput:Nnn \g__str_alias_prop { latin4 } { iso88594 }
\prop_gput:Nnn \g__str_alias_prop { latin5 } { iso88599 }
\prop_gput:Nnn \g__str_alias_prop { latin6 } { iso885910 }
\prop_gput:Nnn \g__str_alias_prop { latin7 } { iso885913 }
\prop_gput:Nnn \g__str_alias_prop { latin8 } { iso885914 }
\prop_gput:Nnn \g__str_alias_prop { latin9 } { iso885915 }
\prop_gput:Nnn \g__str_alias_prop { latin10 } { iso885916 }
\prop_gput:Nnn \g__str_alias_prop { utf16le } { utf16 }
\prop_gput:Nnn \g__str_alias_prop { utf16be } { utf16 }
\prop_gput:Nnn \g__str_alias_prop { utf32le } { utf32 }
\prop_gput:Nnn \g__str_alias_prop { utf32be } { utf32 }
\prop_gput:Nnn \g__str_alias_prop { hexadecimal } { hex }
\bool_lazy_any:nTF
  {
    \sys_if_engine_luatex_p:
    \sys_if_engine_xetex_p:
  }
  {
    \prop_gput:Nnn \g__str_alias_prop { default } {  }
  }
  {
    \prop_gput:Nnn \g__str_alias_prop { default } { utf8 }
  }
```

(*End of definition for* `\g__str_alias_prop`.)

\g__str_error_bool In conversion functions with a built-in conditional, errors are not reported directly to the user, but the information is collected in this boolean, used at the end to decide on which branch of the conditional to take.

```
\bool_new:N \g__str_error_bool
```

(*End of definition for* `\g__str_error_bool`.)

\l__str_byte_flag
\l__str_error_flag Conversions from one ⟨*encoding*⟩/⟨*escaping*⟩ pair to another are done within `e`-expanding assignments. Errors are signalled by raising the relevant flag.

```
\flag_new:N \l__str_byte_flag
\flag_new:N \l__str_error_flag
```

(*End of definition for* `\l__str_byte_flag` *and* `\l__str_error_flag`.)

56.2 String conditionals

__str_if_contains_char:NnT
__str_if_contains_char:NnTF
__str_if_contains_char:nnTF
__str_if_contains_char_aux:nn
__str_if_contains_char_auxi:nN
__str_if_contains_char_true:

`\__str_if_contains_char:nnTF {⟨token list⟩} ⟨char⟩`

Expects the ⟨*token list*⟩ to be an ⟨*other string*⟩: the caller is responsible for ensuring that no (too-)special catcodes remain. Loop over the characters of the string, comparing character codes. The loop is broken if character codes match. Otherwise we return "false".

```
\prg_new_conditional:Npnn \__str_if_contains_char:Nn #1#2 { T , TF }
  {
    \exp_after:wN \__str_if_contains_char_aux:nn \exp_after:wN {#1} {#2}
      { \prg_break:n { ? \fi: } }
    \prg_break_point:
    \prg_return_false:
  }
\cs_new:Npn \__str_if_contains_char_aux:nn #1#2
  { \__str_if_contains_char_auxi:nN {#2} #1 }
\prg_new_conditional:Npnn \__str_if_contains_char:nn #1#2 { TF }
  {
    \__str_if_contains_char_auxi:nN {#2} #1 { \prg_break:n { ? \fi: } }
    \prg_break_point:
    \prg_return_false:
  }
\cs_new:Npn \__str_if_contains_char_auxi:nN #1#2
  {
    \if_charcode:w #1 #2
      \exp_after:wN \__str_if_contains_char_true:
    \fi:
    \__str_if_contains_char_auxi:nN {#1}
  }
\cs_new:Npn \__str_if_contains_char_true:
  { \prg_break:n { \prg_return_true: \use_none:n } }
```

(*End of definition for* `\__str_if_contains_char:NnT` *and others.*)

__str_octal_use:NTF

`\__str_octal_use:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}`

If the ⟨*token*⟩ is an octal digit, it is left in the input stream, *followed* by the ⟨*true code*⟩. Otherwise, the ⟨*false code*⟩ is left in the input stream.

TeXhackers note: This function will fail if the escape character is an octal digit. We are thus careful to set the escape character to a known value before using it. TeX dutifully detects

octal digits for us: if `#1` is an octal digit, then the right-hand side of the comparison is `'1#1`, greater than 1. Otherwise, the right-hand side stops as `'1`, and the conditional takes the `false` branch.

```
\prg_new_conditional:Npnn \__str_octal_use:N #1 { TF }
  {
    \if_int_compare:w 1 < '1 \token_to_str:N #1 \exp_stop_f:
      #1 \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\__str_octal_use:NTF`.)

__str_hexadecimal_use:NTF

TeX detects uppercase hexadecimal digits for us (see `\__str_octal_use:NTF`), but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```
\prg_new_conditional:Npnn \__str_hexadecimal_use:N #1 { TF }
  {
    \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
      #1 \prg_return_true:
    \else:
      \if_case:w \int_eval:n { \exp_after:wN ` \token_to_str:N #1 - `a }
           A
      \or: B
      \or: C
      \or: D
      \or: E
      \or: F
      \else:
        \prg_return_false:
        \exp_after:wN \use_none:n
      \fi:
      \prg_return_true:
    \fi:
  }
```

(*End of definition for* `\__str_hexadecimal_use:NTF`.)

56.3 Conversions

56.3.1 Producing one byte or character

\c__str_byte_0_tl
\c__str_byte_1_tl
\c__str_byte_255_tl
\c__str_byte_-1_tl

For each integer N in the range $[0, 255]$, we create a constant token list which holds three character tokens with category code other: the character with character code N, followed by the representation of N as two hexadecimal digits. The value -1 is given a default token list which ensures that later functions give an empty result for the input -1.

```
\group_begin:
  \__kernel_tl_set:Nx \l__str_internal_tl { \tl_to_str:n { 0123456789ABCDEF } }
  \tl_map_inline:Nn \l__str_internal_tl
    {
      \tl_map_inline:Nn \l__str_internal_tl
        {
          \tl_const:ce { c__str_byte_ \int_eval:n {"#1##1} _tl }
            { \char_generate:nn { "#1##1 } { 12 } #1 ##1 }
        }
    }
\group_end:
\tl_const:cn { c__str_byte_-1_tl } { { } \use_none:n { } }
```

(*End of definition for* `\c__str_byte_0_tl` *and others.*)

__str_output_byte:n
__str_output_byte:w
__str_output_hexadecimal:n
__str_output_end:

Those functions must be used carefully: feeding them a value outside the range $[-1, 255]$ will attempt to use the undefined token list variable `\c__str_byte_`⟨*number*⟩`_tl`. Assuming that the argument is in the right range, we expand the corresponding token list, and pick either the byte (first token) or the hexadecimal representations (second and third tokens). The value -1 produces an empty result in both cases.

```
\cs_new:Npn \__str_output_byte:n #1
  { \__str_output_byte:w #1 \__str_output_end: }
\cs_new:Npn \__str_output_byte:w
  {
    \exp_after:wN \exp_after:wN
    \exp_after:wN \use_i:nnn
    \cs:w c__str_byte_ \int_eval:w
  }
\cs_new:Npn \__str_output_hexadecimal:n #1
  {
    \exp_after:wN \exp_after:wN
    \exp_after:wN \use_none:n
    \cs:w c__str_byte_ \int_eval:n {#1} _tl \cs_end:
  }
\cs_new:Npn \__str_output_end:
  { \scan_stop: _tl \cs_end: }
```

(*End of definition for* `\__str_output_byte:n` *and others.*)

`\__str_output_byte_pair_be:n`
`\__str_output_byte_pair_le:n`
`\__str_output_byte_pair:nnN`

Convert a number in the range $[0, 65535]$ to a pair of bytes, either big-endian or little-endian.

```
\cs_new:Npn \__str_output_byte_pair_be:n #1
  {
    \exp_args:Nf \__str_output_byte_pair:nnN
      { \int_div_truncate:nn { #1 } { "100 } } {#1} \use:nn
  }
\cs_new:Npn \__str_output_byte_pair_le:n #1
  {
    \exp_args:Nf \__str_output_byte_pair:nnN
      { \int_div_truncate:nn { #1 } { "100 } } {#1} \use_ii_i:nn
  }
\cs_new:Npn \__str_output_byte_pair:nnN #1#2#3
  {
    #3
      { \__str_output_byte:n { #1 } }
      { \__str_output_byte:n { #2 - #1 * "100 } }
  }
```

(*End of definition for* `\__str_output_byte_pair_be:n`, `\__str_output_byte_pair_le:n`, *and* `\__str_output_byte_pair:nnN`.)

56.3.2 Mapping functions for conversions

`\__str_convert_gmap:N`
`\__str_convert_gmap_loop:NN`

This maps the function `#1` over all characters in `\g__str_result_tl`, which should be a byte string in most cases, sometimes a native string.

```
\cs_new_protected:Npn \__str_convert_gmap:N #1
  {
    \__kernel_tl_gset:Nx \g__str_result_tl
      {
        \exp_after:wN \__str_convert_gmap_loop:NN
        \exp_after:wN #1
          \g__str_result_tl { ? \prg_break: }
        \prg_break_point:
      }
```

```
  }
\cs_new:Npn \__str_convert_gmap_loop:NN #1#2
  {
    \use_none:n #2
    #1#2
    \__str_convert_gmap_loop:NN #1
  }
```

(*End of definition for* `\__str_convert_gmap:N` *and* `\__str_convert_gmap_loop:NN`.)

`\__str_convert_gmap_internal:N`
`\__str_convert_gmap_internal_loop:Nw`

This maps the function `#1` over all character codes in `\g__str_result_tl`, which must be in the internal representation.

```
\cs_new_protected:Npn \__str_convert_gmap_internal:N #1
  {
    \__kernel_tl_gset:Nx \g__str_result_tl
      {
        \exp_after:wN \__str_convert_gmap_internal_loop:Nww
        \exp_after:wN #1
          \g__str_result_tl \s__str \s__str_stop \prg_break: \s__str
        \prg_break_point:
      }
  }
\cs_new:Npn \__str_convert_gmap_internal_loop:Nww #1 #2 \s__str #3 \s__str
  {
    \__str_use_none_delimit_by_s_stop:w #3 \s__str_stop
    #1 {#3}
    \__str_convert_gmap_internal_loop:Nww #1
  }
```

(*End of definition for* `\__str_convert_gmap_internal:N` *and* `\__str_convert_gmap_internal_loop:Nw`.)

56.3.3 Error-reporting during conversion

`\__str_if_flag_error:Nne`
`\__str_if_flag_no_error:Nne`

When converting using the function `\str_set_convert:Nnnn`, errors should be reported to the user after each step in the conversion. Errors are signalled by raising some flag (typically `@@_error`), so here we test that flag: if it is raised, give the user an error, otherwise remove the arguments. On the other hand, in the conditional functions `\str_set_convert:NnnnTF`, errors should be suppressed. This is done by changing `\__str_if_flag_error:Nne` into `\__str_if_flag_no_error:Nne` locally.

```
\cs_new_protected:Npn \__str_if_flag_error:Nne #1
  {
    \flag_if_raised:NTF #1
      { \msg_error:nne { str } }
      { \use_none:nn }
  }
\cs_new_protected:Npn \__str_if_flag_no_error:Nne #1#2#3
  { \flag_if_raised:NT #1 { \bool_gset_true:N \g__str_error_bool } }
```

(*End of definition for* `\__str_if_flag_error:Nne` *and* `\__str_if_flag_no_error:Nne`.)

`\__str_if_flag_times:NT`

At the end of each conversion step, we raise all relevant errors as one error message, built on the fly. The height of each flag indicates how many times a given error was encountered. This function prints `#2` followed by the number of occurrences of an error if it occurred, nothing otherwise.

```
\cs_new:Npn \__str_if_flag_times:NT #1#2
  { \flag_if_raised:NT #1 { #2~(x \flag_height:N #1 ) } }
```

(*End of definition for* `\__str_if_flag_times:NT`.)

56.3.4 Framework for conversions

Most functions in this module expect to be working with "native" strings. Strings can also be stored as bytes, in one of many encodings, for instance UTF8. The bytes themselves can be expressed in various ways in terms of TEX tokens, for instance as pairs of hexadecimal digits. The questions of going from arbitrary Unicode code points to bytes, and from bytes to tokens are mostly independent.

Conversions are done in four steps:

- "unescape" produces a string of bytes;
- "decode" takes in a string of bytes, and converts it to a list of Unicode characters in an internal representation, with items of the form

  ⟨*bytes*⟩ `\s__str` ⟨*Unicode code point*⟩ `\s__str`

  where we have collected the ⟨*bytes*⟩ which combined to form this particular Unicode character, and the ⟨*Unicode code point*⟩ is in the range [0, "10FFFF].
- "encode" encodes the internal list of code points as a byte string in the new encoding;
- "escape" escapes bytes as requested.

The process is modified in case one of the encoding is empty (or the conversion function has been set equal to the empty encoding because it was not found): then the unescape or escape step is ignored, and the decode or encode steps work on tokens instead of bytes. Otherwise, each step must ensure that it passes a correct byte string or internal string to the next step.

`\str_set_convert:Nnnn`
`\str_gset_convert:Nnnn`
`\str_set_convert:Nnnn`*TF*
`\str_gset_convert:Nnnn`*TF*
`\__str_convert:nNNnnn`

The input string is stored in `\g__str_result_tl`, then we: unescape and decode; encode and escape; exit the group and store the result in the user's variable. The various conversion functions all act on `\g__str_result_tl`. Errors are silenced for the conditional functions by redefining `\__str_if_flag_error:Nne` locally.

```
\cs_new_protected:Npn \str_set_convert:Nnnn
  { \__str_convert:nNNnnn { } \tl_set_eq:NN }
\cs_new_protected:Npn \str_gset_convert:Nnnn
  { \__str_convert:nNNnnn { } \tl_gset_eq:NN }
\prg_new_protected_conditional:Npnn
    \str_set_convert:Nnnn #1#2#3#4 { T , F , TF }
  {
    \bool_gset_false:N \g__str_error_bool
    \__str_convert:nNNnnn
      { \cs_set_eq:NN \__str_if_flag_error:Nne \__str_if_flag_no_error:Nne }
      \tl_set_eq:NN #1 {#2} {#3} {#4}
    \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
  }
\prg_new_protected_conditional:Npnn
    \str_gset_convert:Nnnn #1#2#3#4 { T , F , TF }
```

```
  {
    \bool_gset_false:N \g__str_error_bool
    \__str_convert:nNNnnn
      { \cs_set_eq:NN \__str_if_flag_error:Nne \__str_if_flag_no_error:Nne }
      \tl_gset_eq:NN #1 {#2} {#3} {#4}
    \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
  }
\cs_new_protected:Npn \__str_convert:nNNnnn #1#2#3#4#5#6
  {
    \group_begin:
      #1
      \__kernel_tl_gset:Nx \g__str_result_tl { \__kernel_str_to_other_fast:n {#4} }
      \exp_after:wN \__str_convert:wwwnn
        \tl_to_str:n {#5} /// \s__str_stop
        { decode } { unescape }
        \prg_do_nothing:
        \__str_convert_decode_:
      \exp_after:wN \__str_convert:wwwnn
        \tl_to_str:n {#6} /// \s__str_stop
        { encode } { escape }
        \use_ii_i:nn
        \__str_convert_encode_:
        \__kernel_tl_gset:Nx \g__str_result_tl
          { \tl_to_str:V \g__str_result_tl }
    \group_end:
    #2 #3 \g__str_result_tl
  }
```

(*End of definition for* `\str_set_convert:Nnnn` *and others. These functions are documented on page* *145*.)

`\__str_convert:wwwnn`
`\__str_convert:NNnNN`

The task of `\__str_convert:wwwnn` is to split ⟨*encoding*⟩/⟨*escaping*⟩ pairs into their components, `#1` and `#2`. Calls to `\__str_convert:nnn` ensure that the corresponding conversion functions are defined. The third auxiliary does the main work.

- `#1` is the encoding conversion function;
- `#2` is the escaping function;
- `#3` is the escaping name for use in an error message;
- `#4` is `\prg_do_nothing:` for unescaping/decoding, and `\use_ii_i:nn` for encoding/escaping;
- `#5` is the default encoding function (either "decode" or "encode"), for which there should be no escaping.

Let us ignore the native encoding for a second. In the unescaping/decoding phase, we want to do `#2#1` in this order, and in the encoding/escaping phase, the order should be reversed: `#4#2#1` does exactly that. If one of the encodings is the default (native), then the escaping should be ignored, with an error if any was given, and only the encoding, `#1`, should be performed.

```
\cs_new_protected:Npn \__str_convert:wwwnn
    #1 / #2 // #3 \s__str_stop #4#5
  {
```

```
    \__str_convert:nnn {enc} {#4} {#1}
    \__str_convert:nnn {esc} {#5} {#2}
    \exp_args:Ncc \__str_convert:NNnNN
      { __str_convert_#4_#1: } { __str_convert_#5_#2: } {#2}
  }
\cs_new_protected:Npn \__str_convert:NNnNN #1#2#3#4#5
  {
    \if_meaning:w #1 #5
      \tl_if_empty:nF {#3}
        { \msg_error:nne { str } { native-escaping } {#3} }
      #1
    \else:
      #4 #2 #1
    \fi:
  }
```

(*End of definition for* `\__str_convert:wwwnn` *and* `\__str_convert:NNnNN`.)

`\__str_convert:nnn`
`\__str_convert:nnnn`

The arguments of `\__str_convert:nnn` are: `enc` or `esc`, used to build filenames, the type of the conversion (unescape, decode, encode, escape), and the encoding or escaping name. If the function is already defined, no need to do anything. Otherwise, filter out all non-alphanumerics in the name, and lowercase it. Feed that, and the same three arguments, to `\__str_convert:nnnn`. The task is then to make sure that the conversion function `#3_#1` corresponding to the type `#3` and filtered name `#1` is defined, then set our initial conversion function `#3_#4` equal to that.

How do we get the `#3_#1` conversion to be defined if it isn't? Two main cases.

First, if `#1` is a key in `\g__str_alias_prop`, then the value `\l__str_internal_tl` tells us what file to load. Loading is skipped if the file was already read, *i.e.*, if the conversion command based on `\l__str_internal_tl` already exists. Otherwise, try to load the file; if that fails, there is an error, use the default empty name instead.

Second, `#1` may be absent from the property list. The `\cs_if_exist:cF` test is automatically false, and we search for a file defining the encoding or escaping `#1` (this should allow third-party `.def` files). If the file is not found, there is an error, use the default empty name instead.

In all cases, the conversion based on `\l__str_internal_tl` is defined, so we can set the `#3_#1` function equal to that. In some cases (*e.g.*, `utf16be`), the `#3_#1` function is actually defined within the file we just loaded, and it is different from the `\l__str_internal_tl`-based function: we mustn't clobber that different definition.

```
\cs_new_protected:Npn \__str_convert:nnn #1#2#3
  {
    \cs_if_exist:cF { __str_convert_#2_#3: }
      {
        \exp_args:Ne \__str_convert:nnnn
          { \__str_convert_lowercase_alphanum:n {#3} }
          {#1} {#2} {#3}
      }
  }
\cs_new_protected:Npn \__str_convert:nnnn #1#2#3#4
  {
    \cs_if_exist:cF { __str_convert_#3_#1: }
      {
        \prop_get:NnNF \g__str_alias_prop {#1} \l__str_internal_tl
          { \tl_set:Nn \l__str_internal_tl {#1} }
```

```
        \cs_if_exist:cF { __str_convert_#3_ \l__str_internal_tl : }
          {
            \file_if_exist:nTF { l3str-#2- \l__str_internal_tl .def }
              {
                \group_begin:
                  \cctab_select:N \c_code_cctab
                  \file_input:n { l3str-#2- \l__str_internal_tl .def }
                \group_end:
              }
              {
                \tl_clear:N \l__str_internal_tl
                \msg_error:nnee { str } { unknown-#2 } {#4} {#1}
              }
          }
        \cs_if_exist:cF { __str_convert_#3_#1: }
          {
            \cs_gset_eq:cc { __str_convert_#3_#1: }
              { __str_convert_#3_ \l__str_internal_tl : }
          }
      }
    \cs_gset_eq:cc { __str_convert_#3_#4: } { __str_convert_#3_#1: }
  }
```

(*End of definition for* `\__str_convert:nnn` *and* `\__str_convert:nnnn`.)

`\__str_convert_lowercase_alphanum:n`
`\__str_convert_lowercase_alphanum_loop:N`

This function keeps only letters and digits, with upper case letters converted to lower case.

```
\cs_new:Npn \__str_convert_lowercase_alphanum:n #1
  {
    \exp_after:wN \__str_convert_lowercase_alphanum_loop:N
      \tl_to_str:n {#1} { ? \prg_break: }
    \prg_break_point:
  }
\cs_new:Npn \__str_convert_lowercase_alphanum_loop:N #1
  {
    \use_none:n #1
    \if_int_compare:w `#1 > `Z \exp_stop_f:
      \if_int_compare:w `#1 > `z \exp_stop_f: \else:
        \if_int_compare:w `#1 < `a \exp_stop_f: \else:
          #1
        \fi:
      \fi:
    \else:
      \if_int_compare:w `#1 < `A \exp_stop_f:
        \if_int_compare:w 1 < 1#1 \exp_stop_f:
          #1
        \fi:
      \else:
        \__str_output_byte:n { `#1 + `a - `A }
      \fi:
    \fi:
    \__str_convert_lowercase_alphanum_loop:N
  }
```

(*End of definition for* `\__str_convert_lowercase_alphanum:n` *and* `\__str_convert_lowercase_alphanum_loop:N`.)

56.3.5 Byte unescape and escape

Strings of bytes may need to be stored in auxiliary files in safe "escaping" formats. Each such escaping is only loaded as needed. By default, on input any non-byte is filtered out, while the output simply consists in letting bytes through.

`\__str_filter_bytes:n`
`\__str_filter_bytes_aux:N`

In the case of 8-bit engines, every character is a byte. For Unicode-aware engines, test the character code; non-bytes cause us to raise the flag `\l__str_byte_flag`. Spaces have already been given the correct category code when this function is called.

```
\bool_lazy_any:nTF
  {
    \sys_if_engine_luatex_p:
    \sys_if_engine_xetex_p:
  }
  {
    \cs_new:Npn \__str_filter_bytes:n #1
      {
        \__str_filter_bytes_aux:N #1
          { ? \prg_break: }
        \prg_break_point:
      }
    \cs_new:Npn \__str_filter_bytes_aux:N #1
      {
        \use_none:n #1
        \if_int_compare:w `#1 < 256 \exp_stop_f:
          #1
        \else:
          \flag_raise:N \l__str_byte_flag
        \fi:
        \__str_filter_bytes_aux:N
      }
  }
  { \cs_new_eq:NN \__str_filter_bytes:n \use:n }
```

(*End of definition for* `\__str_filter_bytes:n` *and* `\__str_filter_bytes_aux:N`.)

`\__str_convert_unescape_:`
`\__str_convert_unescape_bytes:`

The simplest unescaping method removes non-bytes from `\g__str_result_tl`.

```
\bool_lazy_any:nTF
  {
    \sys_if_engine_luatex_p:
    \sys_if_engine_xetex_p:
  }
  {
    \cs_new_protected:Npn \__str_convert_unescape_:
      {
        \flag_clear:N \l__str_byte_flag
        \__kernel_tl_gset:Nx \g__str_result_tl
          { \exp_args:No \__str_filter_bytes:n \g__str_result_tl }
        \__str_if_flag_error:Nne \l__str_byte_flag { non-byte } { bytes }
      }
  }
```

```
  { \cs_new_protected:Npn \__str_convert_unescape_: { } }
\cs_new_eq:NN \__str_convert_unescape_bytes: \__str_convert_unescape_:
```

(*End of definition for* `\__str_convert_unescape_:` *and* `\__str_convert_unescape_bytes:`.)

`\__str_convert_escape_:`
`\__str_convert_escape_bytes:`

The simplest form of escape leaves the bytes from the previous step of the conversion unchanged.

```
\cs_new_protected:Npn \__str_convert_escape_: { }
\cs_new_eq:NN \__str_convert_escape_bytes: \__str_convert_escape_:
```

(*End of definition for* `\__str_convert_escape_:` *and* `\__str_convert_escape_bytes:`.)

56.3.6 Native strings

`\__str_convert_decode_:`
`\__str_decode_native_char:N`

Convert each character to its character code, one at a time.

```
\cs_new_protected:Npn \__str_convert_decode_:
  { \__str_convert_gmap:N \__str_decode_native_char:N }
\cs_new:Npn \__str_decode_native_char:N #1
  { #1 \s__str \int_value:w `#1 \s__str }
```

(*End of definition for* `\__str_convert_decode_:` *and* `\__str_decode_native_char:N`.)

`\__str_convert_encode_:`
`\__str_encode_native_char:n`

The conversion from an internal string to native character tokens basically maps `\char_generate:nn` through the code-points, but in non-Unicode-aware engines we use a fallback character `?` rather than nothing when given a character code outside $[0,255]$. We detect the presence of bad characters using a flag and only produce a single error after the `e`-expanding assignment.

```
\bool_lazy_any:nTF
  {
    \sys_if_engine_luatex_p:
    \sys_if_engine_xetex_p:
  }
  {
    \cs_new_protected:Npn \__str_convert_encode_:
      { \__str_convert_gmap_internal:N \__str_encode_native_char:n }
    \cs_new:Npn \__str_encode_native_char:n #1
      { \char_generate:nn {#1} {12} }
  }
  {
    \cs_new_protected:Npn \__str_convert_encode_:
      {
        \flag_clear:N \l__str_error_flag
        \__str_convert_gmap_internal:N \__str_encode_native_char:n
        \__str_if_flag_error:Nne \l__str_error_flag
          { native-overflow } { }
      }
    \cs_new:Npn \__str_encode_native_char:n #1
      {
        \if_int_compare:w #1 > \c__str_max_byte_int
          \flag_raise:N \l__str_error_flag
          ?
        \else:
          \char_generate:nn {#1} {12}
        \fi:
```

```
      }
    \msg_new:nnnn { str } { native-overflow }
      { Character~code~too~large~for~this~engine. }
      {
        This~engine~only~support~8-bit~characters:~
        valid~character~codes~are~in~the~range~[0,255].~
        To~manipulate~arbitrary~Unicode,~use~LuaTeX~or~XeTeX.
      }
  }
```

(*End of definition for* `\__str_convert_encode_:` *and* `\__str_encode_native_char:n`.)

56.3.7 clist

`\__str_convert_decode_clist:`
`\__str_decode_clist_char:n`

Convert each integer to the internal form. We first turn `\g__str_result_tl` into a clist variable, as this avoids problems with leading or trailing commas.

```
\cs_new_protected:Npn \__str_convert_decode_clist:
  {
    \clist_gset:No \g__str_result_tl \g__str_result_tl
    \__kernel_tl_gset:Nx \g__str_result_tl
      {
        \exp_args:No \clist_map_function:nN
          \g__str_result_tl \__str_decode_clist_char:n
      }
  }
\cs_new:Npn \__str_decode_clist_char:n #1
  { #1 \s__str \int_eval:n {#1} \s__str }
```

(*End of definition for* `\__str_convert_decode_clist:` *and* `\__str_decode_clist_char:n`.)

`\__str_convert_encode_clist:`
`\__str_encode_clist_char:n`

Convert the internal list of character codes to a comma-list of character codes. The first line produces a comma-list with a leading comma, removed in the next step (this also works in the empty case, since `\tl_tail:N` does not trigger an error in this case).

```
\cs_new_protected:Npn \__str_convert_encode_clist:
  {
    \__str_convert_gmap_internal:N \__str_encode_clist_char:n
    \__kernel_tl_gset:Nx \g__str_result_tl { \tl_tail:N \g__str_result_tl }
  }
\cs_new:Npn \__str_encode_clist_char:n #1 { , #1 }
```

(*End of definition for* `\__str_convert_encode_clist:` *and* `\__str_encode_clist_char:n`.)

56.3.8 8-bit encodings

It is not clear in what situations 8-bit encodings are used, hence it is not clear what should be optimized. The current approach is reasonably efficient to convert long strings, and it scales well when using many different encodings.

The data needed to support a given 8-bit encoding is stored in a file that consists of a single function call

`\__str_declare_eight_bit_encoding:nnnn {⟨name⟩} {⟨modulo⟩} {⟨mapping⟩} {⟨missing⟩}`

This declares the encoding ⟨*name*⟩ to map bytes to Unicode characters according to the ⟨*mapping*⟩, and map those bytes which are not mentioned in the ⟨*mapping*⟩ either to the replacement character (if they appear in ⟨*missing*⟩), or to themselves. The ⟨*mapping*⟩ argument is a token list of pairs {⟨*byte*⟩} {⟨*Unicode*⟩} expressed in uppercase hexadecimal notation. The ⟨*missing*⟩ argument is a token list of {⟨*byte*⟩}. Every ⟨*byte*⟩ which does not appear in the ⟨*mapping*⟩ nor the ⟨*missing*⟩ lists maps to itself in Unicode, so for instance the `latin1` encoding has empty ⟨*mapping*⟩ and ⟨*missing*⟩ lists. The ⟨*modulo*⟩ is a (decimal) integer between 256 and 558 inclusive, modulo which all Unicode code points supported by the encodings must be different.

We use two integer arrays per encoding. When decoding we only use the `decode` integer array, with entry $n + 1$ (offset needed because integer array indices start at 1) equal to the Unicode code point that corresponds to the n-th byte in the encoding under consideration, or -1 if the given byte is invalid in this encoding. When encoding we use both arrays: upon seeing a code point n, we look up the entry (1 plus) n modulo some number M in the `encode` array, which tells us the byte that might encode the given Unicode code point, then we check in the `decode` array that indeed this byte encodes the Unicode code point we want. Here, M is an encoding-dependent integer between 256 and 558 (it turns out), chosen so that among the Unicode code points that can be validly represented in the given encoding, no pair of code points have the same value modulo M.

__str_declare_eight_bit_encoding:nnnn
__str_declare_eight_bit_aux:NNnnn
__str_declare_eight_bit_loop:Nnn
__str_declare_eight_bit_loop:Nn

Loop through both lists of bytes to fill in the `decode` integer array, then fill the `encode` array accordingly. For bytes that are invalid in the given encoding, store -1 in the `decode` array.

```
\cs_new_protected:Npn \__str_declare_eight_bit_encoding:nnnn #1
  {
    \tl_set:Nn \l__str_internal_tl {#1}
    \cs_new_protected:cpn { __str_convert_decode_#1: }
      { \__str_convert_decode_eight_bit:n {#1} }
    \cs_new_protected:cpn { __str_convert_encode_#1: }
      { \__str_convert_encode_eight_bit:n {#1} }
    \exp_args:Ncc \__str_declare_eight_bit_aux:NNnnn
      { g__str_decode_#1_intarray } { g__str_encode_#1_intarray }
  }
\cs_new_protected:Npn \__str_declare_eight_bit_aux:NNnnn #1#2#3#4#5
  {
    \intarray_new:Nn #1 { 256 }
    \int_step_inline:nnn { 0 } { 255 }
      { \intarray_gset:Nnn #1 { 1 + ##1 } {##1} }
    \__str_declare_eight_bit_loop:Nnn #1
      #4 { \s__str_stop \prg_break: } { }
    \prg_break_point:
    \__str_declare_eight_bit_loop:Nn #1
      #5 { \s__str_stop \prg_break: }
    \prg_break_point:
    \intarray_new:Nn #2 {#3}
    \int_step_inline:nnn { 0 } { 255 }
      {
        \int_compare:nNnF { \intarray_item:Nn #1 { 1 + ##1 } } = { -1 }
          {
            \intarray_gset:Nnn #2
              {
                1 +
```

```
              \int_mod:nn { \intarray_item:Nn #1 { 1 + ##1 } }
                { \intarray_count:N #2 }
            }
            {##1}
        }
    }
  }
\cs_new_protected:Npn \__str_declare_eight_bit_loop:Nnn #1#2#3
  {
    \__str_use_none_delimit_by_s_stop:w #2 \s__str_stop
    \intarray_gset:Nnn #1 { 1 + "#2 } { "#3 }
    \__str_declare_eight_bit_loop:Nnn #1
  }
\cs_new_protected:Npn \__str_declare_eight_bit_loop:Nn #1#2
  {
    \__str_use_none_delimit_by_s_stop:w #2 \s__str_stop
    \intarray_gset:Nnn #1 { 1 + "#2 } { -1 }
    \__str_declare_eight_bit_loop:Nn #1
  }
```

(*End of definition for* `\__str_declare_eight_bit_encoding:nnnn` *and others.*)

__str_convert_decode_eight_bit:n
__str_decode_eight_bit_aux:n
__str_decode_eight_bit_aux:Nn

The map from bytes to Unicode code points is in the `decode` array corresponding to the given encoding. Define `\__str_tmp:w` and pass it successively all bytes in the string. It produces an internal representation with suitable `\s__str` inserted, and the corresponding code point is obtained by looking it up in the integer array. If the entry is -1 then issue a replacement character and raise the flag indicating that there was an error.

```
\cs_new_protected:Npn \__str_convert_decode_eight_bit:n #1
  {
    \cs_set:Npe \__str_tmp:w
      {
        \exp_not:N \__str_decode_eight_bit_aux:Nn
        \exp_not:c { g__str_decode_#1_intarray }
      }
    \flag_clear:N \l__str_error_flag
    \__str_convert_gmap:N \__str_tmp:w
    \__str_if_flag_error:Nne \l__str_error_flag { decode-8-bit } {#1}
  }
\cs_new:Npn \__str_decode_eight_bit_aux:Nn #1#2
  {
    #2 \s__str
    \exp_args:Nf \__str_decode_eight_bit_aux:n
      { \intarray_item:Nn #1 { 1 + `#2 } }
    \s__str
  }
\cs_new:Npn \__str_decode_eight_bit_aux:n #1
  {
    \if_int_compare:w #1 < \c_zero_int
      \flag_raise:N \l__str_error_flag
      \int_value:w \c__str_replacement_char_int
    \else:
      #1
    \fi:
  }
```

(*End of definition for* `\__str_convert_decode_eight_bit:n`*,* `\__str_decode_eight_bit_aux:n`*, and* `\__str_decode_eight_bit_aux:Nn`*.*)

__str_convert_encode_eight_bit:n
__str_encode_eight_bit_aux:nnN
__str_encode_eight_bit_aux:NNn

It is not practical to make an integer array with indices in the full Unicode range, so we work modulo some number, which is simply the size of the `encode` integer array for the given encoding. This gives us a candidate byte for representing a given Unicode code point. Of course taking the modulo leads to collisions so we check in the `decode` array that the byte we got is indeed correct. Otherwise the Unicode code point we started from is simply not representable in the given encoding.

```
\int_new:N \l__str_modulo_int
\cs_new_protected:Npn \__str_convert_encode_eight_bit:n #1
  {
    \cs_set:Npe \__str_tmp:w
      {
        \exp_not:N \__str_encode_eight_bit_aux:NNn
        \exp_not:c { g__str_encode_#1_intarray }
        \exp_not:c { g__str_decode_#1_intarray }
      }
    \flag_clear:N \l__str_error_flag
    \__str_convert_gmap_internal:N \__str_tmp:w
    \__str_if_flag_error:Nne \l__str_error_flag { encode-8-bit } {#1}
  }
\cs_new:Npn \__str_encode_eight_bit_aux:NNn #1#2#3
  {
    \exp_args:Nf \__str_encode_eight_bit_aux:nnN
      {
        \intarray_item:Nn #1
          { 1 + \int_mod:nn {#3} { \intarray_count:N #1 } }
      }
      {#3}
      #2
  }
\cs_new:Npn \__str_encode_eight_bit_aux:nnN #1#2#3
  {
    \int_compare:nNnTF { \intarray_item:Nn #3 { 1 + #1 } } = {#2}
      { \__str_output_byte:n {#1} }
      { \flag_raise:N \l__str_error_flag }
  }
```

(*End of definition for* `\__str_convert_encode_eight_bit:n`*,* `\__str_encode_eight_bit_aux:nnN`*, and* `\__str_encode_eight_bit_aux:NNn`*.*)

56.4 Messages

General messages, and messages for the encodings and escapings loaded by default ("native", and "bytes").

```
\msg_new:nnn { str } { unknown-esc }
  { Escaping~scheme~'#1'~(filtered:~'#2')~unknown. }
\msg_new:nnn { str } { unknown-enc }
  { Encoding~scheme~'#1'~(filtered:~'#2')~unknown. }
\msg_new:nnnn { str } { native-escaping }
  { The~'native'~encoding~scheme~does~not~support~any~escaping. }
  {
```

```
    Since~native~strings~do~not~consist~in~bytes,~
    none~of~the~escaping~methods~make~sense.~
    The~specified~escaping,~'#1',~will~be~ignored.
  }
\msg_new:nnn { str } { file-not-found }
  { File~'l3str-#1.def'~not~found. }
```

Message used when the "bytes" unescaping fails because the string given to `\str_set_convert:Nnnn` contains a non-byte. This cannot happen for the -8-bit engines. Messages used for other escapings and encodings are defined in each definition file.

```
\bool_lazy_any:nT
  {
    \sys_if_engine_luatex_p:
    \sys_if_engine_xetex_p:
  }
  {
    \msg_new:nnnn { str } { non-byte }
      { String~invalid~in~escaping~'#1':~it~may~only~contain~bytes. }
      {
        Some~characters~in~the~string~you~asked~to~convert~are~not~
        8-bit~characters.~Perhaps~the~string~is~a~'native'~Unicode~string?~
        If~it~is,~try~using\\
        \\
        \iow_indent:n
          {
            \iow_char:N\\str_set_convert:Nnnn \\
            \ \ <str~var>~\{~<string>~\}~\{~native~\}~\{~<target~encoding>~\}
          }
      }
  }
```

Those messages are used when converting to and from 8-bit encodings.

```
\msg_new:nnnn { str } { decode-8-bit }
  { Invalid~string~in~encoding~'#1'. }
  {
    LaTeX~came~across~a~byte~which~is~not~defined~to~represent~
    any~character~in~the~encoding~'#1'.
  }
\msg_new:nnnn { str } { encode-8-bit }
  { Unicode~string~cannot~be~converted~to~encoding~'#1'. }
  {
    The~encoding~'#1'~only~contains~a~subset~of~all~Unicode~characters.~
    LaTeX~was~asked~to~convert~a~string~to~that~encoding,~but~that~
    string~contains~a~character~that~'#1'~does~not~support.
  }
```

56.5 Escaping definitions

Several of those encodings are defined by the pdf file format. The following byte storage methods are defined:

- `bytes` (default), non-bytes are filtered out, and bytes are left untouched (this is defined by default);

- hex or hexadecimal, as per the pdfTEX primitive \pdfescapehex
- name, as per the pdfTEX primitive \pdfescapename
- string, as per the pdfTEX primitive \pdfescapestring
- url, as per the percent encoding of urls.

56.5.1 Unescape methods

__str_convert_unescape_hex:
__str_unescape_hex_auxi:N
__str_unescape_hex_auxii:N

Take chars two by two, and interpret each pair as the hexadecimal code for a byte. Anything else than hexadecimal digits is ignored, raising the flag. A string which contains an odd number of hexadecimal digits gets 0 appended to it: this is equivalent to appending a 0 in all cases, and dropping it if it is alone.

```
\cs_new_protected:Npn \__str_convert_unescape_hex:
  {
    \group_begin:
      \flag_clear:N \l__str_error_flag
      \int_set:Nn \tex_escapechar:D { 92 }
      \__kernel_tl_gset:Nx \g__str_result_tl
        {
          \__str_output_byte:w "
            \exp_last_unbraced:Nf \__str_unescape_hex_auxi:N
              { \tl_to_str:N \g__str_result_tl }
            0 { ? 0 - 1 \prg_break: }
            \prg_break_point:
          \__str_output_end:
        }
      \__str_if_flag_error:Nne \l__str_error_flag { unescape-hex } { }
    \group_end:
  }
\cs_new:Npn \__str_unescape_hex_auxi:N #1
  {
    \use_none:n #1
    \__str_hexadecimal_use:NTF #1
      { \__str_unescape_hex_auxii:N }
      {
        \flag_raise:N \l__str_error_flag
        \__str_unescape_hex_auxi:N
      }
  }
\cs_new:Npn \__str_unescape_hex_auxii:N #1
  {
    \use_none:n #1
    \__str_hexadecimal_use:NTF #1
      {
        \__str_output_end:
        \__str_output_byte:w " \__str_unescape_hex_auxi:N
      }
      {
        \flag_raise:N \l__str_error_flag
        \__str_unescape_hex_auxii:N
      }
  }
```

```
\msg_new:nnnn { str } { unescape-hex }
  { String~invalid~in~escaping~'hex':~only~hexadecimal~digits~allowed. }
  {
    Some~characters~in~the~string~you~asked~to~convert~are~not~
    hexadecimal~digits~(0-9,~A-F,~a-f)~nor~spaces.
  }
```

(*End of definition for* `\__str_convert_unescape_hex:`, `\__str_unescape_hex_auxi:N`, *and* `\__str_unescape_hex_auxii:N`.)

`\__str_convert_unescape_name:`
`\__str_unescape_name_loop:wNN`
`\__str_convert_unescape_url:`
`\__str_unescape_url_loop:wNN`

The `\__str_convert_unescape_name:` function replaces each occurrence of `#` followed by two hexadecimal digits in `\g__str_result_tl` by the corresponding byte. The `url` function is identical, with escape character `%` instead of `#`. Thus we define the two together. The arguments of `\__str_tmp:w` are the character code of `#` or `%` in hexadecimal, the name of the main function to define, and the name of the auxiliary which performs the loop.

The looping auxiliary `#3` finds the next escape character, reads the following two characters, and tests them. The test `\__str_hexadecimal_use:NTF` leaves the uppercase digit in the input stream, hence we surround the test with `\__str_output_byte:w "` and `\__str_output_end:`. If both characters are hexadecimal digits, they should be removed before looping: this is done by `\use_i:nnn`. If one of the characters is not a hexadecimal digit, then feed `"#1` to `\__str_output_byte:w` to produce the escape character, raise the flag, and call the looping function followed by the two characters (remove `\use_i:nnn`).

```
\cs_set_protected:Npn \__str_tmp:w #1#2#3
  {
    \cs_new_protected:cpn { __str_convert_unescape_#2: }
      {
        \group_begin:
          \flag_clear:N \l__str_byte_flag
          \flag_clear:N \l__str_error_flag
          \int_set:Nn \tex_escapechar:D { 92 }
          \__kernel_tl_gset:Nx \g__str_result_tl
            {
              \exp_after:wN #3 \g__str_result_tl
                #1 ? { ? \prg_break: }
              \prg_break_point:
            }
          \__str_if_flag_error:Nne \l__str_byte_flag { non-byte } { #2 }
          \__str_if_flag_error:Nne \l__str_error_flag { unescape-#2 } { }
        \group_end:
      }
    \cs_new:Npn #3 ##1#1##2##3
      {
        \__str_filter_bytes:n {##1}
        \use_none:n ##3
        \__str_output_byte:w "
          \__str_hexadecimal_use:NTF ##2
            {
              \__str_hexadecimal_use:NTF ##3
                { }
                {
                  \flag_raise:N \l__str_error_flag
```

```
         * 0 + `#1 \use_i:nn
        }
      }
      {
       \flag_raise:N \l__str_error_flag
       0 + `#1 \use_i:nn
      }
    \__str_output_end:
    \use_i:nnn #3 ##2##3
   }
  \msg_new:nnnn { str } { unescape-#2 }
   { String~invalid~in~escaping~'#2'. }
   {
    LaTeX~came~across~the~escape~character~'#1'~not~followed~by~
    two~hexadecimal~digits.~This~is~invalid~in~the~escaping~'#2'.
   }
 }
\exp_after:wN \__str_tmp:w \c_hash_str { name }
 \__str_unescape_name_loop:wNN
\exp_after:wN \__str_tmp:w \c_percent_str { url }
 \__str_unescape_url_loop:wNN
```

(*End of definition for* `\__str_convert_unescape_name:` *and others.*)

`\__str_convert_unescape_string:`
`\__str_unescape_string_newlines:wN`
`\__str_unescape_string_loop:wNNN`
`\__str_unescape_string_repeat:NNNNNN`

The `string` escaping is somewhat similar to the `name` and `url` escapings, with escape character `\`. The first step is to convert all three line endings, `^^J`, `^^M`, and `^^M^^J` to the common `^^J`, as per the PDF specification. This step cannot raise the flag.

Then the following escape sequences are decoded.

`\n` Line feed (10)

`\r` Carriage return (13)

`\t` Horizontal tab (9)

`\b` Backspace (8)

`\f` Form feed (12)

`\(` Left parenthesis

`\)` Right parenthesis

`\\` Backslash

`\ddd` (backslash followed by 1 to 3 octal digits) Byte `ddd` (octal), subtracting 256 in case of overflow.

If followed by an end-of-line character, the backslash and the end-of-line are ignored. If followed by anything else, the backslash is ignored, raising the error flag.

```
\group_begin:
 \char_set_catcode_other:N \^^J
 \char_set_catcode_other:N \^^M
 \cs_set_protected:Npn \__str_tmp:w #1
  {
   \cs_new_protected:Npn \__str_convert_unescape_string:
```

```
      {
        \group_begin:
          \flag_clear:N \l__str_byte_flag
          \flag_clear:N \l__str_error_flag
          \int_set:Nn \tex_escapechar:D { 92 }
          \__kernel_tl_gset:Nx \g__str_result_tl
            {
              \exp_after:wN \__str_unescape_string_newlines:wN
                \g__str_result_tl \prg_break: ^^M ?
              \prg_break_point:
            }
          \__kernel_tl_gset:Nx \g__str_result_tl
            {
              \exp_after:wN \__str_unescape_string_loop:wNNN
                \g__str_result_tl #1 ?? { ? \prg_break: }
              \prg_break_point:
            }
          \__str_if_flag_error:Nne \l__str_byte_flag { non-byte } { string }
          \__str_if_flag_error:Nne \l__str_error_flag { unescape-string } { }
        \group_end:
      }
  }
\exp_args:No \__str_tmp:w { \c_backslash_str }
\exp_last_unbraced:NNNNo
  \cs_new:Npn \__str_unescape_string_loop:wNNN #1 \c_backslash_str #2#3#4
      {
        \__str_filter_bytes:n {#1}
        \use_none:n #4
        \__str_output_byte:w '
          \__str_octal_use:NTF #2
            {
              \__str_octal_use:NTF #3
                {
                  \__str_octal_use:NTF #4
                    {
                      \if_int_compare:w #2 > 3 \exp_stop_f:
                        - 256
                      \fi:
                      \__str_unescape_string_repeat:NNNNNN
                    }
                    { \__str_unescape_string_repeat:NNNNNN ? }
                }
                { \__str_unescape_string_repeat:NNNNNN ?? }
            }
            {
              \str_case_e:nnF {#2}
                {
                  { \c_backslash_str } { 134 }
                  { ( } { 50 }
                  { ) } { 51 }
                  { r } { 15 }
                  { f } { 14 }
                  { n } { 12 }
                  { t } { 11 }
```

```
                      { b } { 10 }
                      { ^^J } { 0 - 1 }
                    }
                    {
                      \flag_raise:N \l__str_error_flag
                      0 - 1 \use_i:nn
                    }
                }
              \__str_output_end:
              \use_i:nn \__str_unescape_string_loop:wNNN #2#3#4
            }
  \cs_new:Npn \__str_unescape_string_repeat:NNNNNN #1#2#3#4#5#6
    { \__str_output_end: \__str_unescape_string_loop:wNNN }
  \cs_new:Npn \__str_unescape_string_newlines:wN #1 ^^M #2
    {
      #1
      \if_charcode:w ^^J #2 \else: ^^J \fi:
      \__str_unescape_string_newlines:wN #2
    }
  \msg_new:nnnn { str } { unescape-string }
    { String~invalid~in~escaping~'string'. }
    {
      LaTeX~came~across~an~escape~character~'\c_backslash_str'~
      not~followed~by~any~of:~'n',~'r',~'t',~'b',~'f',~'(',~')',~
      '\c_backslash_str',~one~to~three~octal~digits,~or~the~end~
      of~a~line.
    }
\group_end:
```

(*End of definition for* `\__str_convert_unescape_string:` *and others.*)

56.5.2 Escape methods

Currently, none of the escape methods can lead to errors, assuming that their input is made out of bytes.

`\__str_convert_escape_hex:`
`\__str_escape_hex_char:N`

Loop and convert each byte to hexadecimal.

```
\cs_new_protected:Npn \__str_convert_escape_hex:
  { \__str_convert_gmap:N \__str_escape_hex_char:N }
\cs_new:Npn \__str_escape_hex_char:N #1
  { \__str_output_hexadecimal:n { `#1 } }
```

(*End of definition for* `\__str_convert_escape_hex:` *and* `\__str_escape_hex_char:N`*.*)

`\__str_convert_escape_name:`
`\__str_escape_name_char:n`
`\__str_if_escape_name:nTF`
`\c__str_escape_name_str`
`\c__str_escape_name_not_str`

For each byte, test whether it should be output as is, or be "hash-encoded". Roughly, bytes outside the range ["2A, "7E] are hash-encoded. We keep two lists of exceptions: characters in `\c__str_escape_name_not_str` are not hash-encoded, and characters in the `\c__str_escape_name_str` are encoded.

```
\str_const:Nn \c__str_escape_name_not_str { ! " $ & ' } %$
\str_const:Nn \c__str_escape_name_str { {}/<>[] }
\cs_new_protected:Npn \__str_convert_escape_name:
  { \__str_convert_gmap:N \__str_escape_name_char:n }
\cs_new:Npn \__str_escape_name_char:n #1
  {
```

```
    \__str_if_escape_name:nTF {#1} {#1}
      { \c_hash_str \__str_output_hexadecimal:n {`#1} }
  }
\prg_new_conditional:Npnn \__str_if_escape_name:n #1 { TF }
  {
    \if_int_compare:w `#1 < "2A \exp_stop_f:
      \__str_if_contains_char:NnTF \c__str_escape_name_not_str {#1}
        \prg_return_true: \prg_return_false:
    \else:
      \if_int_compare:w `#1 > "7E \exp_stop_f:
        \prg_return_false:
      \else:
        \__str_if_contains_char:NnTF \c__str_escape_name_str {#1}
          \prg_return_false: \prg_return_true:
      \fi:
    \fi:
  }
```

(*End of definition for* `\__str_convert_escape_name:` *and others.*)

__str_convert_escape_string:
__str_escape_string_char:N
__str_if_escape_string:NTF
\c__str_escape_string_str

Any character below (and including) space, and any character above (and including) `del`, are converted to octal. One backslash is added before each parenthesis and backslash.

```
\str_const:Ne \c__str_escape_string_str
  { \c_backslash_str ( ) }
\cs_new_protected:Npn \__str_convert_escape_string:
  { \__str_convert_gmap:N \__str_escape_string_char:N }
\cs_new:Npn \__str_escape_string_char:N #1
  {
    \__str_if_escape_string:NTF #1
      {
        \__str_if_contains_char:NnT
          \c__str_escape_string_str {#1}
          { \c_backslash_str }
        #1
      }
      {
        \c_backslash_str
        \int_div_truncate:nn {`#1} {64}
        \int_mod:nn { \int_div_truncate:nn {`#1} { 8 } } { 8 }
        \int_mod:nn {`#1} { 8 }
      }
  }
\prg_new_conditional:Npnn \__str_if_escape_string:N #1 { TF }
  {
    \if_int_compare:w `#1 < "27 \exp_stop_f:
      \prg_return_false:
    \else:
      \if_int_compare:w `#1 > "7A \exp_stop_f:
        \prg_return_false:
      \else:
        \prg_return_true:
      \fi:
    \fi:
  }
```

(*End of definition for* `\__str_convert_escape_string:` *and others.*)

`\__str_convert_escape_url:`
`\__str_escape_url_char:n`
`\__str_if_escape_url:nTF`

This function is similar to `\__str_convert_escape_name:`, escaping different characters.

```
\cs_new_protected:Npn \__str_convert_escape_url:
  { \__str_convert_gmap:N \__str_escape_url_char:n }
\cs_new:Npn \__str_escape_url_char:n #1
  {
    \__str_if_escape_url:nTF {#1} {#1}
      { \c_percent_str \__str_output_hexadecimal:n { `#1 } }
  }
\prg_new_conditional:Npnn \__str_if_escape_url:n #1 { TF }
  {
    \if_int_compare:w `#1 < "30 \exp_stop_f:
      \__str_if_contains_char:nnTF { "-. } {#1}
        \prg_return_true: \prg_return_false:
    \else:
      \if_int_compare:w `#1 > "7E \exp_stop_f:
        \prg_return_false:
      \else:
        \__str_if_contains_char:nnTF { : ; = ? @ [ ] } {#1}
          \prg_return_false: \prg_return_true:
      \fi:
    \fi:
  }
```

(*End of definition for* `\__str_convert_escape_url:`, `\__str_escape_url_char:n`, *and* `\__str_if_escape_url:nTF`.)

56.6 Encoding definitions

The `native` encoding is automatically defined. Other encodings are loaded as needed. The following encodings are supported:

- UTF-8;
- UTF-16, big-, little-endian, or with byte order mark;
- UTF-32, big-, little-endian, or with byte order mark;
- the ISO 8859 code pages, numbered from 1 to 16, skipping the inexistent ISO 8859-12.

56.6.1 utf-8 support

`\__str_convert_encode_utf8:`
`\__str_encode_utf_viii_char:n`
`\__str_encode_utf_viii_loop:wwnnw`

Loop through the internal string, and convert each character to its UTF-8 representation. The representation is built from the right-most (least significant) byte to the left-most (most significant) byte. Continuation bytes are in the range $[128, 191]$, taking 64 different values, hence we roughly want to express the character code in base 64, shifting the first digit in the representation by some number depending on how many continuation bytes there are. In the range $[0, 127]$, output the corresponding byte directly. In the range $[128, 2047]$, output the remainder modulo 64, plus 128 as a continuation byte, then output the quotient (which is in the range $[0, 31]$), shifted by 192. In the next range, $[2048, 65535]$, split the character code into residue and quotient modulo 64, output the residue as a first continuation byte, then repeat; this leaves us with a quotient in the range $[0, 15]$, which we output shifted by 224. The last range, $[65536, 1114111]$, follows the same pattern: once we realize that dividing twice by 64 leaves us with a number larger than 15, we repeat, producing a last continuation byte, and offset the quotient by 240 for the leading byte.

How is that implemented? `\__str_encode_utf_vii_loop:wwnnw` takes successive quotients as its first argument, the quotient from the previous step as its second argument (except in step 1), the bound for quotients that trigger one more step or not, and finally the offset used if this step should produce the leading byte. Leading bytes can be in the ranges $[0, 127]$, $[192, 223]$, $[224, 239]$, and $[240, 247]$ (really, that last limit should be 244 because Unicode stops at the code point 1114111). At each step, if the quotient `#1` is less than the limit `#3` for that range, output the leading byte (`#1` shifted by `#4`) and stop. Otherwise, we need one more step: use the quotient of `#1` by 64, and `#1` as arguments for the looping auxiliary, and output the continuation byte corresponding to the remainder $\#2 - 64\#1 + 128$. The bizarre construction `- 1 + 0 *` removes the spurious initial continuation byte (better methods welcome).

```
\cs_new_protected:cpn { __str_convert_encode_utf8: }
  { \__str_convert_gmap_internal:N \__str_encode_utf_viii_char:n }
\cs_new:Npn \__str_encode_utf_viii_char:n #1
  {
    \__str_encode_utf_viii_loop:wwnnw #1 ; - 1 + 0 * ;
      { 128 } {       0 }
      {  32 } {     192 }
      {  16 } {     224 }
      {   8 } {     240 }
    \s__str_stop
  }
\cs_new:Npn \__str_encode_utf_viii_loop:wwnnw #1; #2; #3#4 #5 \s__str_stop
  {
    \if_int_compare:w #1 < #3 \exp_stop_f:
      \__str_output_byte:n { #1 + #4 }
      \exp_after:wN \__str_use_none_delimit_by_s_stop:w
    \fi:
    \exp_after:wN \__str_encode_utf_viii_loop:wwnnw
      \int_value:w \int_div_truncate:nn {#1} {64} ; #1 ;
      #5 \s__str_stop
    \__str_output_byte:n { #2 - 64 * ( #1 - 2 ) }
  }
```

(*End of definition for* `\__str_convert_encode_utf8:` *,* `\__str_encode_utf_viii_char:n` *, and* `\__str_encode_utf_viii_loop:wwnnw`*.*)

`__str_missing`
`__str_extra`
`__str_overlong`
`__str_overflow`

When decoding a string that is purportedly in the UTF-8 encoding, four different errors can occur, signalled by a specific flag for each (we define those flags using `\flag_clear_new:N` rather than `\flag_new:N`, because they are shared with other encoding definition files).

- "Missing continuation byte": a leading byte is not followed by the right number of continuation bytes.
- "Extra continuation byte": a continuation byte appears where it was not expected, *i.e.*, not after an appropriate leading byte.

- "Overlong": a Unicode character is expressed using more bytes than necessary, for instance, "C0"80 for the code point 0, instead of a single null byte.
- "Overflow": this occurs when decoding produces Unicode code points greater than 1114111.

We only raise one LaTeX3 error message, combining all the errors which occurred. In the short message, the leading comma must be removed to get a grammatically correct sentence. In the long text, first remind the user what a correct UTF-8 string should look like, then add error-specific information.

```
\flag_clear_new:N \l__str_missing_flag
\flag_clear_new:N \l__str_extra_flag
\flag_clear_new:N \l__str_overlong_flag
\flag_clear_new:N \l__str_overflow_flag
\msg_new:nnnn { str } { utf8-decode }
  {
    Invalid~UTF-8~string:
    \exp_last_unbraced:Nf \use_none:n
      {
        \__str_if_flag_times:NT \l__str_missing_flag  { ,~missing~continuation~byte }
        \__str_if_flag_times:NT \l__str_extra_flag    { ,~extra~continuation~byte }
        \__str_if_flag_times:NT \l__str_overlong_flag { ,~overlong~form }
        \__str_if_flag_times:NT \l__str_overflow_flag { ,~code~point~too~large }
      }
    .
  }
  {
    In~the~UTF-8~encoding,~each~Unicode~character~consists~in~
    1~to~4~bytes,~with~the~following~bit~pattern: \\
    \iow_indent:n
      {
        Code~point~\ \ \ \ <~128:~0xxxxxxx \\
        Code~point~\ \ \  <~2048:~110xxxxx~10xxxxxx \\
        Code~point~\ \   <~65536:~1110xxxx~10xxxxxx~10xxxxxx \\
        Code~point~    <~1114112:~11110xxx~10xxxxxx~10xxxxxx~10xxxxxx \\
      }
    Bytes~of~the~form~10xxxxxx~are~called~continuation~bytes.
    \flag_if_raised:NT \l__str_missing_flag
      {
        \\\\
        A~leading~byte~(in~the~range~[192,255])~was~not~followed~by~
        the~appropriate~number~of~continuation~bytes.
      }
    \flag_if_raised:NT \l__str_extra_flag
      {
        \\\\
        LaTeX~came~across~a~continuation~byte~when~it~was~not~expected.
      }
    \flag_if_raised:NT \l__str_overlong_flag
      {
        \\\\
        Every~Unicode~code~point~must~be~expressed~in~the~shortest~
        possible~form.~For~instance,~'0xC0'~'0x83'~is~not~a~valid~
        representation~for~the~code~point~3.
```

```
      }
    \flag_if_raised:NT \l__str_overflow_flag
      {
        \\\\
        Unicode~limits~code~points~to~the~range~[0,1114111].
      }
  }
\prop_gput:Nnn \g_msg_module_name_prop { str } { LaTeX }
\prop_gput:Nnn \g_msg_module_type_prop { str } { }
```

(*End of definition for* `__str_missing` *and others.*)

`\__str_convert_decode_utf8:`
`\__str_decode_utf_viii_start:N`
`\__str_decode_utf_viii_continuation:wwN`
`\__str_decode_utf_viii_aux:wNnnwN`
`\__str_decode_utf_viii_overflow:w`
`\__str_decode_utf_viii_end:`

Decoding is significantly harder than encoding. As before, lower some flags, which are tested at the end (in bulk, to trigger at most one LaTeX3 error, as explained above). We expect successive multi-byte sequences of the form ⟨*start byte*⟩ ⟨*continuation bytes*⟩. The `_start` auxiliary tests the first byte:

- [0, "7F]: the byte stands alone, and is converted to its own character code;
- ["80, "BF]: unexpected continuation byte, raise the appropriate flag, and convert that byte to the replacement character "FFFD;
- ["C0, "FF]: this byte should be followed by some continuation byte(s).

In the first two cases, `\use_none_delimit_by_q_stop:w` removes data that only the third case requires, namely the limits of ranges of Unicode characters which can be expressed with 1, 2, 3, or 4 bytes.

We can now concentrate on the multi-byte case and the `_continuation` auxiliary. We expect `#3` to be in the range ["80, "BF]. The test for this goes as follows: if the character code is less than "80, we compare it to −"C0, yielding `false`; otherwise to "C0, yielding `true` in the range ["80, "BF] and `false` otherwise. If we find that the byte is not a continuation range, stop the current slew of bytes, output the replacement character, and continue parsing with the `_start` auxiliary, starting at the byte we just tested. Once we know that the byte is a continuation byte, leave it behind us in the input stream, compute what code point the bytes read so far would produce, and feed that number to the `_aux` function.

The `_aux` function tests whether we should look for more continuation bytes or not. If the number it receives as `#1` is less than the maximum `#4` for the current range, then we are done: check for an overlong representation by comparing `#1` with the maximum `#3` for the previous range. Otherwise, we call the `_continuation` auxiliary again, after shifting the "current code point" by `#4` (maximum from the range we just checked).

Two additional tests are needed: if we reach the end of the list of range maxima and we are still not done, then we are faced with an overflow. Clean up, and again insert the code point "FFFD for the replacement character. Also, every time we read a byte, we need to check whether we reached the end of the string. In a correct UTF-8 string, this happens automatically when the `_start` auxiliary leaves its first argument in the input stream: the end-marker begins with `\prg_break:`, which ends the loop. On the other hand, if the end is reached when looking for a continuation byte, the `\use_none:n #3` construction removes the first token from the end-marker, and leaves the `_end` auxiliary, which raises the appropriate error flag before ending the mapping.

```
\cs_new_protected:cpn { __str_convert_decode_utf8: }
  {
```

```
    \flag_clear:N \l__str_error_flag
    \flag_clear:N \l__str_missing_flag
    \flag_clear:N \l__str_extra_flag
    \flag_clear:N \l__str_overlong_flag
    \flag_clear:N \l__str_overflow_flag
    \__kernel_tl_gset:Nx \g__str_result_tl
      {
        \exp_after:wN \__str_decode_utf_viii_start:N \g__str_result_tl
          { \prg_break: \__str_decode_utf_viii_end: }
        \prg_break_point:
      }
    \__str_if_flag_error:Nne \l__str_error_flag { utf8-decode } { }
  }
\cs_new:Npn \__str_decode_utf_viii_start:N #1
  {
    #1
    \if_int_compare:w `#1 < "C0 \exp_stop_f:
      \s__str
      \if_int_compare:w `#1 < "80 \exp_stop_f:
        \int_value:w `#1
      \else:
        \flag_raise:N \l__str_extra_flag
        \flag_raise:N \l__str_error_flag
        \int_use:N \c__str_replacement_char_int
      \fi:
    \else:
      \exp_after:wN \__str_decode_utf_viii_continuation:wwN
      \int_value:w \int_eval:n { `#1 - "C0 } \exp_after:wN
    \fi:
    \s__str
    \__str_use_none_delimit_by_s_stop:w {"80} {"800} {"10000} {"110000} \s__str_stop
    \__str_decode_utf_viii_start:N
  }
\cs_new:Npn \__str_decode_utf_viii_continuation:wwN
    #1 \s__str #2 \__str_decode_utf_viii_start:N #3
  {
    \use_none:n #3
    \if_int_compare:w `#3 <
          \if_int_compare:w `#3 < "80 \exp_stop_f: - \fi:
          "C0 \exp_stop_f:
      #3
      \exp_after:wN \__str_decode_utf_viii_aux:wNnnwN
      \int_value:w \int_eval:n { #1 * "40 + `#3 - "80 } \exp_after:wN
    \else:
      \s__str
      \flag_raise:N \l__str_missing_flag
      \flag_raise:N \l__str_error_flag
      \int_use:N \c__str_replacement_char_int
    \fi:
    \s__str
    #2
    \__str_decode_utf_viii_start:N #3
  }
\cs_new:Npn \__str_decode_utf_viii_aux:wNnnwN
```

```
    #1 \s__str #2#3#4 #5 \__str_decode_utf_viii_start:N #6
  {
    \if_int_compare:w #1 < #4 \exp_stop_f:
      \s__str
      \if_int_compare:w #1 < #3 \exp_stop_f:
        \flag_raise:N \l__str_overlong_flag
        \flag_raise:N \l__str_error_flag
        \int_use:N \c__str_replacement_char_int
      \else:
        #1
      \fi:
    \else:
      \if_meaning:w \s__str_stop #5
        \__str_decode_utf_viii_overflow:w #1
      \fi:
      \exp_after:wN \__str_decode_utf_viii_continuation:wwN
      \int_value:w \int_eval:n { #1 - #4 } \exp_after:wN
    \fi:
    \s__str
    #2 {#4} #5
    \__str_decode_utf_viii_start:N
  }
\cs_new:Npn \__str_decode_utf_viii_overflow:w #1 \fi: #2 \fi:
  {
    \fi: \fi:
    \flag_raise:N \l__str_overflow_flag
    \flag_raise:N \l__str_error_flag
    \int_use:N \c__str_replacement_char_int
  }
\cs_new:Npn \__str_decode_utf_viii_end:
  {
    \s__str
    \flag_raise:N \l__str_missing_flag
    \flag_raise:N \l__str_error_flag
    \int_use:N \c__str_replacement_char_int \s__str
    \prg_break:
  }
```

(*End of definition for* `\__str_convert_decode_utf8:` *and others.*)

56.6.2 utf-16 support

The definitions are done in a category code regime where the bytes 254 and 255 used by the byte order mark have catcode 12.

```
\group_begin:
  \char_set_catcode_other:N \^^fe
  \char_set_catcode_other:N \^^ff
```

`\__str_convert_encode_utf16:`
`\__str_convert_encode_utf16be:`
`\__str_convert_encode_utf16le:`
`\__str_encode_utf_xvi_aux:N`
`\__str_encode_utf_xvi_char:n`

When the endianness is not specified, it is big-endian by default, and we add a byte-order mark. Convert characters one by one in a loop, with different behaviours depending on the character code.

- [0, "D7FF]: converted to two bytes;

- ["D800, "DFFF] are used as surrogates: they cannot be converted and are replaced by the replacement character;
- ["E000, "FFFF]: converted to two bytes;
- ["10000, "10FFFF]: converted to a pair of surrogates, each two bytes. The magic "D7C0 is "D800 − "10000/"400.

For the duration of this operation, `\__str_tmp:w` is defined as a function to convert a number in the range [0, "FFFF] to a pair of bytes (either big endian or little endian), by feeding the quotient of the division of `#1` by "100, followed by `#1` to `\__str_encode_-utf_xvi_be:nn` or its `le` analog: those compute the remainder, and output two bytes for the quotient and remainder.

```
\cs_new_protected:cpn { __str_convert_encode_utf16: }
  {
    \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n
    \tl_gput_left:Ne \g__str_result_tl { ^^fe ^^ff }
  }
\cs_new_protected:cpn { __str_convert_encode_utf16be: }
  { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n }
\cs_new_protected:cpn { __str_convert_encode_utf16le: }
  { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_le:n }
\cs_new_protected:Npn \__str_encode_utf_xvi_aux:N #1
  {
    \flag_clear:N \l__str_error_flag
    \cs_set_eq:NN \__str_tmp:w #1
    \__str_convert_gmap_internal:N \__str_encode_utf_xvi_char:n
    \__str_if_flag_error:Nne \l__str_error_flag { utf16-encode } { }
  }
\cs_new:Npn \__str_encode_utf_xvi_char:n #1
  {
    \if_int_compare:w #1 < "D800 \exp_stop_f:
      \__str_tmp:w {#1}
    \else:
      \if_int_compare:w #1 < "10000 \exp_stop_f:
        \if_int_compare:w #1 < "E000 \exp_stop_f:
          \flag_raise:N \l__str_error_flag
          \__str_tmp:w { \c__str_replacement_char_int }
        \else:
          \__str_tmp:w {#1}
        \fi:
      \else:
        \exp_args:Nf \__str_tmp:w { \int_div_truncate:nn {#1} {"400} + "D7C0 }
        \exp_args:Nf \__str_tmp:w { \int_mod:nn {#1} {"400} + "DC00 }
      \fi:
    \fi:
  }
```

(*End of definition for* `\__str_convert_encode_utf16:` *and others.*)

`__str_missing` `__str_extra` `__str_end` When encoding a Unicode string to UTF-16, only one error can occur: code points in the range ["D800, "DFFF], corresponding to surrogates, cannot be encoded. We use the all-purpose flag `@@_error` to signal that error.

When decoding a Unicode string which is purportedly in UTF-16, three errors can occur: a missing trail surrogate, an unexpected trail surrogate, and a string containing an odd number of bytes.

```
\flag_clear_new:N \l__str_missing_flag
\flag_clear_new:N \l__str_extra_flag
\flag_clear_new:N \l__str_end_flag
\msg_new:nnnn { str } { utf16-encode }
  { Unicode~string~cannot~be~expressed~in~UTF-16:~surrogate. }
  {
    Surrogate~code~points~(in~the~range~[U+D800,~U+DFFF])~
    can~be~expressed~in~the~UTF-8~and~UTF-32~encodings,~
    but~not~in~the~UTF-16~encoding.
  }
\msg_new:nnnn { str } { utf16-decode }
  {
    Invalid~UTF-16~string:
    \exp_last_unbraced:Nf \use_none:n
      {
        \__str_if_flag_times:NT \l__str_missing_flag  { ,~missing~trail~surrogate }
        \__str_if_flag_times:NT \l__str_extra_flag    { ,~extra~trail~surrogate }
        \__str_if_flag_times:NT \l__str_end_flag      { ,~odd~number~of~bytes }
      }
    .
  }
  {
    In~the~UTF-16~encoding,~each~Unicode~character~is~encoded~as~
    2~or~4~bytes: \\
    \iow_indent:n
      {
        Code~point~in~[U+0000,~U+D7FF]:~two~bytes \\
        Code~point~in~[U+D800,~U+DFFF]:~illegal \\
        Code~point~in~[U+E000,~U+FFFF]:~two~bytes \\
        Code~point~in~[U+10000,~U+10FFFF]:~
          a~lead~surrogate~and~a~trail~surrogate \\
      }
    Lead~surrogates~are~pairs~of~bytes~in~the~range~[0xD800,~0xDBFF],~
    and~trail~surrogates~are~in~the~range~[0xDC00,~0xDFFF].
    \flag_if_raised:NT \l__str_missing_flag
      {
        \\\\
        A~lead~surrogate~was~not~followed~by~a~trail~surrogate.
      }
    \flag_if_raised:NT \l__str_extra_flag
      {
        \\\\
        LaTeX~came~across~a~trail~surrogate~when~it~was~not~expected.
      }
    \flag_if_raised:NT \l__str_end_flag
      {
        \\\\
        The~string~contained~an~odd~number~of~bytes.~This~is~invalid:~
        the~basic~code~unit~for~UTF-16~is~16~bits~(2~bytes).
      }
  }
```

(*End of definition for* `__str_missing`*,* `__str_extra`*, and* `__str_end`*.*)

`\__str_convert_decode_utf16:`
`\__str_convert_decode_utf16be:`
`\__str_convert_decode_utf16le:`
`\__str_decode_utf_xvi_bom:NN`
`\__str_decode_utf_xvi:Nw`

As for UTF-8, decoding UTF-16 is harder than encoding it. If the endianness is unknown, check the first two bytes: if those are "FE and "FF in either order, remove them and use the corresponding endianness, otherwise assume big-endianness. The three endianness cases are based on a common auxiliary whose first argument is 1 for big-endian and 2 for little-endian, and whose second argument, delimited by the scan mark `\s__str_stop`, is expanded once (the string may be long; passing `\g__str_result_tl` as an argument before expansion is cheaper).

The `\__str_decode_utf_xvi:Nw` function defines `\__str_tmp:w` to take two arguments and return the character code of the first one if the string is big-endian, and the second one if the string is little-endian, then loops over the string using `\__str_decode_utf_xvi_pair:NN` described below.

```
\cs_new_protected:cpn { __str_convert_decode_utf16be: }
  { \__str_decode_utf_xvi:Nw 1 \g__str_result_tl \s__str_stop }
\cs_new_protected:cpn { __str_convert_decode_utf16le: }
  { \__str_decode_utf_xvi:Nw 2 \g__str_result_tl \s__str_stop }
\cs_new_protected:cpn { __str_convert_decode_utf16: }
  {
    \exp_after:wN \__str_decode_utf_xvi_bom:NN
      \g__str_result_tl \s__str_stop \s__str_stop \s__str_stop
  }
\cs_new_protected:Npn \__str_decode_utf_xvi_bom:NN #1#2
  {
    \str_if_eq:nnTF { #1#2 } { ^^ff ^^fe }
      { \__str_decode_utf_xvi:Nw 2 }
      {
        \str_if_eq:nnTF { #1#2 } { ^^fe ^^ff }
          { \__str_decode_utf_xvi:Nw 1 }
          { \__str_decode_utf_xvi:Nw 1 #1#2 }
      }
  }
\cs_new_protected:Npn \__str_decode_utf_xvi:Nw #1#2 \s__str_stop
  {
    \flag_clear:N \l__str_error_flag
    \flag_clear:N \l__str_missing_flag
    \flag_clear:N \l__str_extra_flag
    \flag_clear:N \l__str_end_flag
    \cs_set:Npn \__str_tmp:w ##1 ##2 { ` ## #1 }
    \__kernel_tl_gset:Nx \g__str_result_tl
      {
        \exp_after:wN \__str_decode_utf_xvi_pair:NN
          #2 \q__str_nil \q__str_nil
        \prg_break_point:
      }
    \__str_if_flag_error:Nne \l__str_error_flag { utf16-decode } { }
  }
```

(*End of definition for* `\__str_convert_decode_utf16:` *and others.*)

`\__str_decode_utf_xvi_pair:NN`
`\__str_decode_utf_xvi_quad:NNwNN`
`\__str_decode_utf_xvi_pair_end:Nw`
`\__str_decode_utf_xvi_error:nNN`
`\__str_decode_utf_xvi_extra:NNw`

Bytes are read two at a time. At this stage, `\@@_tmp:w #1#2` expands to the character code of the most significant byte, and we distinguish cases depending on which range it lies in:

- ["D8, "DB] signals a lead surrogate, and the integer expression yields 1 (ε-TeX rounds ties away from zero);
- ["DC, "DF] signals a trail surrogate, unexpected here, and the integer expression yields 2;
- any other value signals a code point in the Basic Multilingual Plane, which stands for itself, and the `\if_case:w` construction expands to nothing (cases other than 1 or 2), leaving the relevant material in the input stream, followed by another call to the `_pair` auxiliary.

The case of a lead surrogate is treated by the `_quad` auxiliary, whose arguments `#1`, `#2`, `#4` and `#5` are the four bytes. We expect the most significant byte of `#4#5` to be in the range ["DC, "DF] (trail surrogate). The test is similar to the test used for continuation bytes in the UTF-8 decoding functions. In the case where `#4#5` is indeed a trail surrogate, leave `#1#2#4#5 \s__str` ⟨*code point*⟩ `\s__str`, and remove the pair `#4#5` before looping with `\__str_decode_utf_xvi_pair:NN`. Otherwise, of course, complain about the missing surrogate.

The magic number "D7F7 is such that "D7F7∗"400 = "D800∗"400+"DC00−"10000.

Every time we read a pair of bytes, we test for the end-marker `\q__str_nil`. When reaching the end, we additionally check that the string had an even length. Also, if the end is reached when expecting a trail surrogate, we treat that as a missing surrogate.

```
\cs_new:Npn \__str_decode_utf_xvi_pair:NN #1#2
  {
    \if_meaning:w \q__str_nil #2
      \__str_decode_utf_xvi_pair_end:Nw #1
    \fi:
    \if_case:w
      \int_eval:n { ( \__str_tmp:w #1#2 - "D6 ) / 4 } \scan_stop:
    \or: \exp_after:wN \__str_decode_utf_xvi_quad:NNwNN
    \or: \exp_after:wN \__str_decode_utf_xvi_extra:NNw
    \fi:
    #1#2 \s__str
    \int_eval:n { "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 } \s__str
    \__str_decode_utf_xvi_pair:NN
  }
\cs_new:Npn \__str_decode_utf_xvi_quad:NNwNN
    #1#2 #3 \__str_decode_utf_xvi_pair:NN #4#5
  {
    \if_meaning:w \q__str_nil #5
      \__str_decode_utf_xvi_error:nNN { missing } #1#2
      \__str_decode_utf_xvi_pair_end:Nw #4
    \fi:
    \if_int_compare:w
        \if_int_compare:w \__str_tmp:w #4#5 < "DC \exp_stop_f:
          0 = 1
        \else:
          \__str_tmp:w #4#5 < "E0
        \fi:
        \exp_stop_f:
      #1 #2 #4 #5 \s__str
      \int_eval:n
        {
```

```
          ( "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 - "D7F7 ) * "400
          + "100 * \__str_tmp:w #4#5 + \__str_tmp:w #5#4
        }
      \s__str
      \exp_after:wN \use_i:nnn
    \else:
      \__str_decode_utf_xvi_error:nNN { missing } #1#2
    \fi:
    \__str_decode_utf_xvi_pair:NN #4#5
  }
\cs_new:Npn \__str_decode_utf_xvi_pair_end:Nw #1 \fi:
  {
    \fi:
    \if_meaning:w \q__str_nil #1
    \else:
      \__str_decode_utf_xvi_error:nNN { end } #1 \prg_do_nothing:
    \fi:
    \prg_break:
  }
\cs_new:Npn \__str_decode_utf_xvi_extra:NNw #1#2 \s__str #3 \s__str
  { \__str_decode_utf_xvi_error:nNN { extra } #1#2 }
\cs_new:Npn \__str_decode_utf_xvi_error:nNN #1#2#3
  {
    \flag_raise:N \l__str_error_flag
    \flag_raise:c { l__str_#1_flag }
    #2 #3 \s__str
    \int_use:N \c__str_replacement_char_int \s__str
  }
```

(*End of definition for* `\__str_decode_utf_xvi_pair:NN` *and others.*)

Restore the original catcodes of bytes 254 and 255.

```
\group_end:
```

56.6.3 utf-32 support

The definitions are done in a category code regime where the bytes 0, 254 and 255 used by the byte order mark have catcode "other".

```
\group_begin:
  \char_set_catcode_other:N \^^00
  \char_set_catcode_other:N \^^fe
  \char_set_catcode_other:N \^^ff
```

`\__str_convert_encode_utf32:`
`\__str_convert_encode_utf32be:`
`\__str_convert_encode_utf32le:`
`\__str_encode_utf_xxxii_be:n`
`\__str_encode_utf_xxxii_be_aux:nn`
`\__str_encode_utf_xxxii_le:n`
`\__str_encode_utf_xxxii_le_aux:nn`

Convert each integer in the comma-list `\g__str_result_tl` to a sequence of four bytes. The functions for big-endian and little-endian encodings are very similar, but the `\__str_output_byte:n` instructions are reversed.

```
  \cs_new_protected:cpn { __str_convert_encode_utf32: }
    {
      \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_be:n
      \tl_gput_left:Ne \g__str_result_tl { ^^00 ^^00 ^^fe ^^ff }
    }
  \cs_new_protected:cpn { __str_convert_encode_utf32be: }
    { \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_be:n }
  \cs_new_protected:cpn { __str_convert_encode_utf32le: }
```

```
  { \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_le:n }
\cs_new:Npn \__str_encode_utf_xxxii_be:n #1
  {
    \exp_args:Nf \__str_encode_utf_xxxii_be_aux:nn
      { \int_div_truncate:nn {#1} { "100 } } {#1}
  }
\cs_new:Npn \__str_encode_utf_xxxii_be_aux:nn #1#2
  {
    ^^00
    \__str_output_byte_pair_be:n {#1}
    \__str_output_byte:n { #2 - #1 * "100 }
  }
\cs_new:Npn \__str_encode_utf_xxxii_le:n #1
  {
    \exp_args:Nf \__str_encode_utf_xxxii_le_aux:nn
      { \int_div_truncate:nn {#1} { "100 } } {#1}
  }
\cs_new:Npn \__str_encode_utf_xxxii_le_aux:nn #1#2
  {
    \__str_output_byte:n { #2 - #1 * "100 }
    \__str_output_byte_pair_le:n {#1}
    ^^00
  }
```

(*End of definition for* `\__str_convert_encode_utf32:` *and others.*)

`\__str_overflow`
`\__str_end`

There can be no error when encoding in UTF-32. When decoding, the string may not have length $4n$, or it may contain code points larger than "10FFFF. The latter case often happens if the encoding was in fact not UTF-32, because most arbitrary strings are not valid in UTF-32.

```
\flag_clear_new:N \l__str_overflow_flag
\flag_clear_new:N \l__str_end_flag
\msg_new:nnnn { str } { utf32-decode }
  {
    Invalid~UTF-32~string:
    \exp_last_unbraced:Nf \use_none:n
      {
        \__str_if_flag_times:NT \l__str_overflow_flag { ,~code~point~too~large }
        \__str_if_flag_times:NT \l__str_end_flag      { ,~truncated~string }
      }
    .
  }
  {
    In~the~UTF-32~encoding,~every~Unicode~character~
    (in~the~range~[U+0000,~U+10FFFF])~is~encoded~as~4~bytes.
    \flag_if_raised:NT \l__str_overflow_flag
      {
        \\\\
        LaTeX~came~across~a~code~point~larger~than~1114111,~
        the~maximum~code~point~defined~by~Unicode.~
        Perhaps~the~string~was~not~encoded~in~the~UTF-32~encoding?
      }
    \flag_if_raised:NT \l__str_end_flag
      {
```

```
          \\\\
          The~length~of~the~string~is~not~a~multiple~of~4.~
          Perhaps~the~string~was~truncated?
        }
    }
```

(*End of definition for* `__str_overflow` *and* `__str_end`.)

`\__str_convert_decode_utf32:`
`\__str_convert_decode_utf32be:`
`\__str_convert_decode_utf32le:`
`\__str_decode_utf_xxxii_bom:NNNN`
`\__str_decode_utf_xxxii:Nw`
`\__str_decode_utf_xxxii_loop:NNNN`
`\__str_decode_utf_xxxii_end:w`

The structure is similar to UTF-16 decoding functions. If the endianness is not given, test the first 4 bytes of the string (possibly `\s__str_stop` if the string is too short) for the presence of a byte-order mark. If there is a byte-order mark, use that endianness, and remove the 4 bytes, otherwise default to big-endian, and leave the 4 bytes in place. The `\__str_decode_utf_xxxii:Nw` auxiliary receives 1 or 2 as its first argument indicating endianness, and the string to convert as its second argument (expanded or not). It sets `\__str_tmp:w` to expand to the character code of either of its two arguments depending on endianness, then triggers the `_loop` auxiliary inside an `e`-expanding assignment to `\g__str_result_tl`.

The `_loop` auxiliary first checks for the end-of-string marker `\s__str_stop`, calling the `_end` auxiliary if appropriate. Otherwise, leave the ⟨*4 bytes*⟩ `\s__str` behind, then check that the code point is not overflowing: the leading byte must be 0, and the following byte at most 16.

In the ending code, we check that there remains no byte: there should be nothing left until the first `\s__str_stop`. Break the map.

```
  \cs_new_protected:cpn { __str_convert_decode_utf32be: }
    { \__str_decode_utf_xxxii:Nw 1 \g__str_result_tl \s__str_stop }
  \cs_new_protected:cpn { __str_convert_decode_utf32le: }
    { \__str_decode_utf_xxxii:Nw 2 \g__str_result_tl \s__str_stop }
  \cs_new_protected:cpn { __str_convert_decode_utf32: }
    {
      \exp_after:wN \__str_decode_utf_xxxii_bom:NNNN \g__str_result_tl
        \s__str_stop \s__str_stop \s__str_stop \s__str_stop \s__str_stop
    }
  \cs_new_protected:Npn \__str_decode_utf_xxxii_bom:NNNN #1#2#3#4
    {
      \str_if_eq:nnTF { #1#2#3#4 } { ^^ff ^^fe ^^00 ^^00 }
        { \__str_decode_utf_xxxii:Nw 2 }
        {
          \str_if_eq:nnTF { #1#2#3#4 } { ^^00 ^^00 ^^fe ^^ff }
            { \__str_decode_utf_xxxii:Nw 1 }
            { \__str_decode_utf_xxxii:Nw 1 #1#2#3#4 }
        }
    }
  \cs_new_protected:Npn \__str_decode_utf_xxxii:Nw #1#2 \s__str_stop
    {
      \flag_clear:N \l__str_overflow_flag
      \flag_clear:N \l__str_end_flag
      \flag_clear:N \l__str_error_flag
      \cs_set:Npn \__str_tmp:w ##1 ##2 { ` ## #1 }
      \__kernel_tl_gset:Nx \g__str_result_tl
        {
          \exp_after:wN \__str_decode_utf_xxxii_loop:NNNN
            #2 \s__str_stop \s__str_stop \s__str_stop \s__str_stop
          \prg_break_point:
```

```
        }
      \__str_if_flag_error:Nne \l__str_error_flag { utf32-decode } { }
    }
  \cs_new:Npn \__str_decode_utf_xxxii_loop:NNNN #1#2#3#4
    {
      \if_meaning:w \s__str_stop #4
        \exp_after:wN \__str_decode_utf_xxxii_end:w
      \fi:
      #1#2#3#4 \s__str
      \if_int_compare:w \__str_tmp:w #1#4 > \c_zero_int
        \flag_raise:N \l__str_overflow_flag
        \flag_raise:N \l__str_error_flag
        \int_use:N \c__str_replacement_char_int
      \else:
        \if_int_compare:w \__str_tmp:w #2#3 > 16 \exp_stop_f:
          \flag_raise:N \l__str_overflow_flag
          \flag_raise:N \l__str_error_flag
          \int_use:N \c__str_replacement_char_int
        \else:
          \int_eval:n
            { \__str_tmp:w #2#3*"10000 + \__str_tmp:w #3#2*"100 + \__str_tmp:w #4#1 }
        \fi:
      \fi:
      \s__str
      \__str_decode_utf_xxxii_loop:NNNN
    }
  \cs_new:Npn \__str_decode_utf_xxxii_end:w #1 \s__str_stop
    {
      \tl_if_empty:nF {#1}
        {
          \flag_raise:N \l__str_end_flag
          \flag_raise:N \l__str_error_flag
          #1 \s__str
          \int_use:N \c__str_replacement_char_int \s__str
        }
      \prg_break:
    }
```

(*End of definition for* `\__str_convert_decode_utf32:` *and others.*)

Restore the original catcodes of bytes 0, 254 and 255.

```
\group_end:
```

56.7 PDF names and strings by expansion

`\str_convert_pdfname:n`
`\__str_convert_pdfname:n`
`\__str_convert_pdfname_bytes:n`
`\__str_convert_pdfname_bytes_aux:n`
`\__str_convert_pdfname_bytes_aux:nnn`

To convert to PDF names by expansion, we work purely on UTF-8 input. The first step is to make a string with "other" spaces, after which we use a simple token-by-token approach. In Unicode engines, we break down everything before one-byte codepoints, but for 8-bit engines there is no need to worry. Actual escaping is covered by the same code as used in the non-expandable route.

```
\cs_new:Npn \str_convert_pdfname:n #1
  {
    \exp_args:Ne \tl_to_str:n
```

```
      { \str_map_function:nN {#1} \__str_convert_pdfname:n }
  }
\bool_lazy_or:nnTF
  { \sys_if_engine_luatex_p: }
  { \sys_if_engine_xetex_p: }
  {
    \cs_new:Npn \__str_convert_pdfname:n #1
      {
        \int_compare:nNnTF { `#1 } > { "7F }
          { \__str_convert_pdfname_bytes:n {#1} }
          { \__str_escape_name_char:n {#1} }
      }
    \cs_new:Npn \__str_convert_pdfname_bytes:n #1
      {
        \exp_args:Ne \__str_convert_pdfname_bytes_aux:n
          { \__kernel_codepoint_to_bytes:n {`#1} }
      }
    \cs_new:Npn \__str_convert_pdfname_bytes_aux:n #1
      { \__str_convert_pdfname_bytes_aux:nnnn #1 }
    \cs_new:Npe \__str_convert_pdfname_bytes_aux:nnnn #1#2#3#4
      {
        \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#1}
        \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#2}
        \exp_not:N \tl_if_blank:nF {#3}
          {
            \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#3}
            \exp_not:N \tl_if_blank:nF {#4}
              {
                \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#4}
              }
          }
      }
  }
  { \cs_new_eq:NN \__str_convert_pdfname:n \__str_escape_name_char:n }
```

(*End of definition for* `\str_convert_pdfname:n` *and others. This function is documented on page* *145*.)

```
</package>
```

56.7.1 iso 8859 support

The ISO-8859-1 encoding exactly matches with the 256 first Unicode characters. For other 8-bit encodings of the ISO-8859 family, we keep track only of differences, and of unassigned bytes.

```
<*iso88591>
\__str_declare_eight_bit_encoding:nnnn { iso88591 } { 256 }
  {
  }
  {
  }
</iso88591>
<*iso88592>
\__str_declare_eight_bit_encoding:nnnn { iso88592 } { 399 }
  {
```

```
{ A1 } { 0104 }
{ A2 } { 02D8 }
{ A3 } { 0141 }
{ A5 } { 013D }
{ A6 } { 015A }
{ A9 } { 0160 }
{ AA } { 015E }
{ AB } { 0164 }
{ AC } { 0179 }
{ AE } { 017D }
{ AF } { 017B }
{ B1 } { 0105 }
{ B2 } { 02DB }
{ B3 } { 0142 }
{ B5 } { 013E }
{ B6 } { 015B }
{ B7 } { 02C7 }
{ B9 } { 0161 }
{ BA } { 015F }
{ BB } { 0165 }
{ BC } { 017A }
{ BD } { 02DD }
{ BE } { 017E }
{ BF } { 017C }
{ C0 } { 0154 }
{ C3 } { 0102 }
{ C5 } { 0139 }
{ C6 } { 0106 }
{ C8 } { 010C }
{ CA } { 0118 }
{ CC } { 011A }
{ CF } { 010E }
{ D0 } { 0110 }
{ D1 } { 0143 }
{ D2 } { 0147 }
{ D5 } { 0150 }
{ D8 } { 0158 }
{ D9 } { 016E }
{ DB } { 0170 }
{ DE } { 0162 }
{ E0 } { 0155 }
{ E3 } { 0103 }
{ E5 } { 013A }
{ E6 } { 0107 }
{ E8 } { 010D }
{ EA } { 0119 }
{ EC } { 011B }
{ EF } { 010F }
{ F0 } { 0111 }
{ F1 } { 0144 }
{ F2 } { 0148 }
{ F5 } { 0151 }
{ F8 } { 0159 }
{ F9 } { 016F }
```

```
    { FB } { 0171 }
    { FE } { 0163 }
    { FF } { 02D9 }
  }
  {
  }
⟨/iso88592⟩
⟨*iso88593⟩
\__str_declare_eight_bit_encoding:nnnn { iso88593 } { 384 }
  {
    { A1 } { 0126 }
    { A2 } { 02D8 }
    { A6 } { 0124 }
    { A9 } { 0130 }
    { AA } { 015E }
    { AB } { 011E }
    { AC } { 0134 }
    { AF } { 017B }
    { B1 } { 0127 }
    { B6 } { 0125 }
    { B9 } { 0131 }
    { BA } { 015F }
    { BB } { 011F }
    { BC } { 0135 }
    { BF } { 017C }
    { C5 } { 010A }
    { C6 } { 0108 }
    { D5 } { 0120 }
    { D8 } { 011C }
    { DD } { 016C }
    { DE } { 015C }
    { E5 } { 010B }
    { E6 } { 0109 }
    { F5 } { 0121 }
    { F8 } { 011D }
    { FD } { 016D }
    { FE } { 015D }
    { FF } { 02D9 }
  }
  {
    { A5 }
    { AE }
    { BE }
    { C3 }
    { D0 }
    { E3 }
    { F0 }
  }
⟨/iso88593⟩
⟨*iso88594⟩
\__str_declare_eight_bit_encoding:nnnn { iso88594 } { 383 }
  {
    { A1 } { 0104 }
```

```
      { A2 } { 0138 }
      { A3 } { 0156 }
      { A5 } { 0128 }
      { A6 } { 013B }
      { A9 } { 0160 }
      { AA } { 0112 }
      { AB } { 0122 }
      { AC } { 0166 }
      { AE } { 017D }
      { B1 } { 0105 }
      { B2 } { 02DB }
      { B3 } { 0157 }
      { B5 } { 0129 }
      { B6 } { 013C }
      { B7 } { 02C7 }
      { B9 } { 0161 }
      { BA } { 0113 }
      { BB } { 0123 }
      { BC } { 0167 }
      { BD } { 014A }
      { BE } { 017E }
      { BF } { 014B }
      { C0 } { 0100 }
      { C7 } { 012E }
      { C8 } { 010C }
      { CA } { 0118 }
      { CC } { 0116 }
      { CF } { 012A }
      { D0 } { 0110 }
      { D1 } { 0145 }
      { D2 } { 014C }
      { D3 } { 0136 }
      { D9 } { 0172 }
      { DD } { 0168 }
      { DE } { 016A }
      { E0 } { 0101 }
      { E7 } { 012F }
      { E8 } { 010D }
      { EA } { 0119 }
      { EC } { 0117 }
      { EF } { 012B }
      { F0 } { 0111 }
      { F1 } { 0146 }
      { F2 } { 014D }
      { F3 } { 0137 }
      { F9 } { 0173 }
      { FD } { 0169 }
      { FE } { 016B }
      { FF } { 02D9 }
    }
    {
    }
⟨/iso88594⟩
⟨*iso88595⟩
```

```
\__str_declare_eight_bit_encoding:nnnn { iso88595 } { 374 }
  {
    { A1 } { 0401 }
    { A2 } { 0402 }
    { A3 } { 0403 }
    { A4 } { 0404 }
    { A5 } { 0405 }
    { A6 } { 0406 }
    { A7 } { 0407 }
    { A8 } { 0408 }
    { A9 } { 0409 }
    { AA } { 040A }
    { AB } { 040B }
    { AC } { 040C }
    { AE } { 040E }
    { AF } { 040F }
    { B0 } { 0410 }
    { B1 } { 0411 }
    { B2 } { 0412 }
    { B3 } { 0413 }
    { B4 } { 0414 }
    { B5 } { 0415 }
    { B6 } { 0416 }
    { B7 } { 0417 }
    { B8 } { 0418 }
    { B9 } { 0419 }
    { BA } { 041A }
    { BB } { 041B }
    { BC } { 041C }
    { BD } { 041D }
    { BE } { 041E }
    { BF } { 041F }
    { C0 } { 0420 }
    { C1 } { 0421 }
    { C2 } { 0422 }
    { C3 } { 0423 }
    { C4 } { 0424 }
    { C5 } { 0425 }
    { C6 } { 0426 }
    { C7 } { 0427 }
    { C8 } { 0428 }
    { C9 } { 0429 }
    { CA } { 042A }
    { CB } { 042B }
    { CC } { 042C }
    { CD } { 042D }
    { CE } { 042E }
    { CF } { 042F }
    { D0 } { 0430 }
    { D1 } { 0431 }
    { D2 } { 0432 }
    { D3 } { 0433 }
    { D4 } { 0434 }
    { D5 } { 0435 }
```

```
    { D6 } { 0436 }
    { D7 } { 0437 }
    { D8 } { 0438 }
    { D9 } { 0439 }
    { DA } { 043A }
    { DB } { 043B }
    { DC } { 043C }
    { DD } { 043D }
    { DE } { 043E }
    { DF } { 043F }
    { E0 } { 0440 }
    { E1 } { 0441 }
    { E2 } { 0442 }
    { E3 } { 0443 }
    { E4 } { 0444 }
    { E5 } { 0445 }
    { E6 } { 0446 }
    { E7 } { 0447 }
    { E8 } { 0448 }
    { E9 } { 0449 }
    { EA } { 044A }
    { EB } { 044B }
    { EC } { 044C }
    { ED } { 044D }
    { EE } { 044E }
    { EF } { 044F }
    { F0 } { 2116 }
    { F1 } { 0451 }
    { F2 } { 0452 }
    { F3 } { 0453 }
    { F4 } { 0454 }
    { F5 } { 0455 }
    { F6 } { 0456 }
    { F7 } { 0457 }
    { F8 } { 0458 }
    { F9 } { 0459 }
    { FA } { 045A }
    { FB } { 045B }
    { FC } { 045C }
    { FD } { 00A7 }
    { FE } { 045E }
    { FF } { 045F }
  }
  {
  }
⟨/iso88595⟩
⟨*iso88596⟩
\__str_declare_eight_bit_encoding:nnnn { iso88596 } { 344 }
  {
    { AC } { 060C }
    { BB } { 061B }
    { BF } { 061F }
    { C1 } { 0621 }
    { C2 } { 0622 }
```

```
    { C3 } { 0623 }
    { C4 } { 0624 }
    { C5 } { 0625 }
    { C6 } { 0626 }
    { C7 } { 0627 }
    { C8 } { 0628 }
    { C9 } { 0629 }
    { CA } { 062A }
    { CB } { 062B }
    { CC } { 062C }
    { CD } { 062D }
    { CE } { 062E }
    { CF } { 062F }
    { D0 } { 0630 }
    { D1 } { 0631 }
    { D2 } { 0632 }
    { D3 } { 0633 }
    { D4 } { 0634 }
    { D5 } { 0635 }
    { D6 } { 0636 }
    { D7 } { 0637 }
    { D8 } { 0638 }
    { D9 } { 0639 }
    { DA } { 063A }
    { E0 } { 0640 }
    { E1 } { 0641 }
    { E2 } { 0642 }
    { E3 } { 0643 }
    { E4 } { 0644 }
    { E5 } { 0645 }
    { E6 } { 0646 }
    { E7 } { 0647 }
    { E8 } { 0648 }
    { E9 } { 0649 }
    { EA } { 064A }
    { EB } { 064B }
    { EC } { 064C }
    { ED } { 064D }
    { EE } { 064E }
    { EF } { 064F }
    { F0 } { 0650 }
    { F1 } { 0651 }
    { F2 } { 0652 }
  }
  {
    { A1 }
    { A2 }
    { A3 }
    { A5 }
    { A6 }
    { A7 }
    { A8 }
    { A9 }
    { AA }
```

```
    { AB }
    { AE }
    { AF }
    { B0 }
    { B1 }
    { B2 }
    { B3 }
    { B4 }
    { B5 }
    { B6 }
    { B7 }
    { B8 }
    { B9 }
    { BA }
    { BC }
    { BD }
    { BE }
    { C0 }
    { DB }
    { DC }
    { DD }
    { DE }
    { DF }
  }
```
⟨/iso88596⟩

⟨∗iso88597⟩
```
\__str_declare_eight_bit_encoding:nnnn { iso88597 } { 498 }
  {
    { A1 } { 2018 }
    { A2 } { 2019 }
    { A4 } { 20AC }
    { A5 } { 20AF }
    { AA } { 037A }
    { AF } { 2015 }
    { B4 } { 0384 }
    { B5 } { 0385 }
    { B6 } { 0386 }
    { B8 } { 0388 }
    { B9 } { 0389 }
    { BA } { 038A }
    { BC } { 038C }
    { BE } { 038E }
    { BF } { 038F }
    { C0 } { 0390 }
    { C1 } { 0391 }
    { C2 } { 0392 }
    { C3 } { 0393 }
    { C4 } { 0394 }
    { C5 } { 0395 }
    { C6 } { 0396 }
    { C7 } { 0397 }
    { C8 } { 0398 }
    { C9 } { 0399 }
    { CA } { 039A }
```

```
    { CB } { 039B }
    { CC } { 039C }
    { CD } { 039D }
    { CE } { 039E }
    { CF } { 039F }
    { D0 } { 03A0 }
    { D1 } { 03A1 }
    { D3 } { 03A3 }
    { D4 } { 03A4 }
    { D5 } { 03A5 }
    { D6 } { 03A6 }
    { D7 } { 03A7 }
    { D8 } { 03A8 }
    { D9 } { 03A9 }
    { DA } { 03AA }
    { DB } { 03AB }
    { DC } { 03AC }
    { DD } { 03AD }
    { DE } { 03AE }
    { DF } { 03AF }
    { E0 } { 03B0 }
    { E1 } { 03B1 }
    { E2 } { 03B2 }
    { E3 } { 03B3 }
    { E4 } { 03B4 }
    { E5 } { 03B5 }
    { E6 } { 03B6 }
    { E7 } { 03B7 }
    { E8 } { 03B8 }
    { E9 } { 03B9 }
    { EA } { 03BA }
    { EB } { 03BB }
    { EC } { 03BC }
    { ED } { 03BD }
    { EE } { 03BE }
    { EF } { 03BF }
    { F0 } { 03C0 }
    { F1 } { 03C1 }
    { F2 } { 03C2 }
    { F3 } { 03C3 }
    { F4 } { 03C4 }
    { F5 } { 03C5 }
    { F6 } { 03C6 }
    { F7 } { 03C7 }
    { F8 } { 03C8 }
    { F9 } { 03C9 }
    { FA } { 03CA }
    { FB } { 03CB }
    { FC } { 03CC }
    { FD } { 03CD }
    { FE } { 03CE }
  }
  {
    { AE }
```

```
    { D2 }
  }
⟨/iso88597⟩
⟨*iso88598⟩
\__str_declare_eight_bit_encoding:nnnn { iso88598 } { 308 }
  {
    { AA } { 00D7 }
    { BA } { 00F7 }
    { DF } { 2017 }
    { E0 } { 05D0 }
    { E1 } { 05D1 }
    { E2 } { 05D2 }
    { E3 } { 05D3 }
    { E4 } { 05D4 }
    { E5 } { 05D5 }
    { E6 } { 05D6 }
    { E7 } { 05D7 }
    { E8 } { 05D8 }
    { E9 } { 05D9 }
    { EA } { 05DA }
    { EB } { 05DB }
    { EC } { 05DC }
    { ED } { 05DD }
    { EE } { 05DE }
    { EF } { 05DF }
    { F0 } { 05E0 }
    { F1 } { 05E1 }
    { F2 } { 05E2 }
    { F3 } { 05E3 }
    { F4 } { 05E4 }
    { F5 } { 05E5 }
    { F6 } { 05E6 }
    { F7 } { 05E7 }
    { F8 } { 05E8 }
    { F9 } { 05E9 }
    { FA } { 05EA }
    { FD } { 200E }
    { FE } { 200F }
  }
  {
    { A1 }
    { BF }
    { C0 }
    { C1 }
    { C2 }
    { C3 }
    { C4 }
    { C5 }
    { C6 }
    { C7 }
    { C8 }
    { C9 }
    { CA }
    { CB }
```

```
    { CC }
    { CD }
    { CE }
    { CF }
    { D0 }
    { D1 }
    { D2 }
    { D3 }
    { D4 }
    { D5 }
    { D6 }
    { D7 }
    { D8 }
    { D9 }
    { DA }
    { DB }
    { DC }
    { DD }
    { DE }
    { FB }
    { FC }
  }
⟨/iso88598⟩
⟨*iso88599⟩
\__str_declare_eight_bit_encoding:nnnn { iso88599 } { 352 }
  {
    { D0 } { 011E }
    { DD } { 0130 }
    { DE } { 015E }
    { F0 } { 011F }
    { FD } { 0131 }
    { FE } { 015F }
  }
  {
  }
⟨/iso88599⟩
⟨*iso885910⟩
\__str_declare_eight_bit_encoding:nnnn { iso885910 } { 383 }
  {
    { A1 } { 0104 }
    { A2 } { 0112 }
    { A3 } { 0122 }
    { A4 } { 012A }
    { A5 } { 0128 }
    { A6 } { 0136 }
    { A8 } { 013B }
    { A9 } { 0110 }
    { AA } { 0160 }
    { AB } { 0166 }
    { AC } { 017D }
    { AE } { 016A }
    { AF } { 014A }
    { B1 } { 0105 }
```

```
    { B2 } { 0113 }
    { B3 } { 0123 }
    { B4 } { 012B }
    { B5 } { 0129 }
    { B6 } { 0137 }
    { B8 } { 013C }
    { B9 } { 0111 }
    { BA } { 0161 }
    { BB } { 0167 }
    { BC } { 017E }
    { BD } { 2015 }
    { BE } { 016B }
    { BF } { 014B }
    { C0 } { 0100 }
    { C7 } { 012E }
    { C8 } { 010C }
    { CA } { 0118 }
    { CC } { 0116 }
    { D1 } { 0145 }
    { D2 } { 014C }
    { D7 } { 0168 }
    { D9 } { 0172 }
    { E0 } { 0101 }
    { E7 } { 012F }
    { E8 } { 010D }
    { EA } { 0119 }
    { EC } { 0117 }
    { F1 } { 0146 }
    { F2 } { 014D }
    { F7 } { 0169 }
    { F9 } { 0173 }
    { FF } { 0138 }
  }
  {
  }
⟨/iso885910⟩
⟨*iso885911⟩
\__str_declare_eight_bit_encoding:nnnn { iso885911 } { 369 }
  {
    { A1 } { 0E01 }
    { A2 } { 0E02 }
    { A3 } { 0E03 }
    { A4 } { 0E04 }
    { A5 } { 0E05 }
    { A6 } { 0E06 }
    { A7 } { 0E07 }
    { A8 } { 0E08 }
    { A9 } { 0E09 }
    { AA } { 0E0A }
    { AB } { 0E0B }
    { AC } { 0E0C }
    { AD } { 0E0D }
    { AE } { 0E0E }
    { AF } { 0E0F }
```

```
{ B0 } { 0E10 }
{ B1 } { 0E11 }
{ B2 } { 0E12 }
{ B3 } { 0E13 }
{ B4 } { 0E14 }
{ B5 } { 0E15 }
{ B6 } { 0E16 }
{ B7 } { 0E17 }
{ B8 } { 0E18 }
{ B9 } { 0E19 }
{ BA } { 0E1A }
{ BB } { 0E1B }
{ BC } { 0E1C }
{ BD } { 0E1D }
{ BE } { 0E1E }
{ BF } { 0E1F }
{ C0 } { 0E20 }
{ C1 } { 0E21 }
{ C2 } { 0E22 }
{ C3 } { 0E23 }
{ C4 } { 0E24 }
{ C5 } { 0E25 }
{ C6 } { 0E26 }
{ C7 } { 0E27 }
{ C8 } { 0E28 }
{ C9 } { 0E29 }
{ CA } { 0E2A }
{ CB } { 0E2B }
{ CC } { 0E2C }
{ CD } { 0E2D }
{ CE } { 0E2E }
{ CF } { 0E2F }
{ D0 } { 0E30 }
{ D1 } { 0E31 }
{ D2 } { 0E32 }
{ D3 } { 0E33 }
{ D4 } { 0E34 }
{ D5 } { 0E35 }
{ D6 } { 0E36 }
{ D7 } { 0E37 }
{ D8 } { 0E38 }
{ D9 } { 0E39 }
{ DA } { 0E3A }
{ DF } { 0E3F }
{ E0 } { 0E40 }
{ E1 } { 0E41 }
{ E2 } { 0E42 }
{ E3 } { 0E43 }
{ E4 } { 0E44 }
{ E5 } { 0E45 }
{ E6 } { 0E46 }
{ E7 } { 0E47 }
{ E8 } { 0E48 }
{ E9 } { 0E49 }
```

```
    { EA } { 0E4A }
    { EB } { 0E4B }
    { EC } { 0E4C }
    { ED } { 0E4D }
    { EE } { 0E4E }
    { EF } { 0E4F }
    { F0 } { 0E50 }
    { F1 } { 0E51 }
    { F2 } { 0E52 }
    { F3 } { 0E53 }
    { F4 } { 0E54 }
    { F5 } { 0E55 }
    { F6 } { 0E56 }
    { F7 } { 0E57 }
    { F8 } { 0E58 }
    { F9 } { 0E59 }
    { FA } { 0E5A }
    { FB } { 0E5B }
  }
  {
    { DB }
    { DC }
    { DD }
    { DE }
  }
⟨/iso885911⟩
⟨*iso885913⟩
\__str_declare_eight_bit_encoding:nnnn { iso885913 } { 399 }
  {
    { A1 } { 201D }
    { A5 } { 201E }
    { A8 } { 00D8 }
    { AA } { 0156 }
    { AF } { 00C6 }
    { B4 } { 201C }
    { B8 } { 00F8 }
    { BA } { 0157 }
    { BF } { 00E6 }
    { C0 } { 0104 }
    { C1 } { 012E }
    { C2 } { 0100 }
    { C3 } { 0106 }
    { C6 } { 0118 }
    { C7 } { 0112 }
    { C8 } { 010C }
    { CA } { 0179 }
    { CB } { 0116 }
    { CC } { 0122 }
    { CD } { 0136 }
    { CE } { 012A }
    { CF } { 013B }
    { D0 } { 0160 }
    { D1 } { 0143 }
    { D2 } { 0145 }
```

```
    { D4 } { 014C }
    { D8 } { 0172 }
    { D9 } { 0141 }
    { DA } { 015A }
    { DB } { 016A }
    { DD } { 017B }
    { DE } { 017D }
    { E0 } { 0105 }
    { E1 } { 012F }
    { E2 } { 0101 }
    { E3 } { 0107 }
    { E6 } { 0119 }
    { E7 } { 0113 }
    { E8 } { 010D }
    { EA } { 017A }
    { EB } { 0117 }
    { EC } { 0123 }
    { ED } { 0137 }
    { EE } { 012B }
    { EF } { 013C }
    { F0 } { 0161 }
    { F1 } { 0144 }
    { F2 } { 0146 }
    { F4 } { 014D }
    { F8 } { 0173 }
    { F9 } { 0142 }
    { FA } { 015B }
    { FB } { 016B }
    { FD } { 017C }
    { FE } { 017E }
    { FF } { 2019 }
  }
  {
  }
⟨/iso885913⟩
⟨*iso885914⟩
\__str_declare_eight_bit_encoding:nnnn { iso885914 } { 529 }
  {
    { A1 } { 1E02 }
    { A2 } { 1E03 }
    { A4 } { 010A }
    { A5 } { 010B }
    { A6 } { 1E0A }
    { A8 } { 1E80 }
    { AA } { 1E82 }
    { AB } { 1E0B }
    { AC } { 1EF2 }
    { AF } { 0178 }
    { B0 } { 1E1E }
    { B1 } { 1E1F }
    { B2 } { 0120 }
    { B3 } { 0121 }
    { B4 } { 1E40 }
    { B5 } { 1E41 }
```

```
    { B7 } { 1E56 }
    { B8 } { 1E81 }
    { B9 } { 1E57 }
    { BA } { 1E83 }
    { BB } { 1E60 }
    { BC } { 1EF3 }
    { BD } { 1E84 }
    { BE } { 1E85 }
    { BF } { 1E61 }
    { D0 } { 0174 }
    { D7 } { 1E6A }
    { DE } { 0176 }
    { F0 } { 0175 }
    { F7 } { 1E6B }
    { FE } { 0177 }
  }
  {
  }
⟨/iso885914⟩
⟨*iso885915⟩
\__str_declare_eight_bit_encoding:nnnn { iso885915 } { 383 }
  {
    { A4 } { 20AC }
    { A6 } { 0160 }
    { A8 } { 0161 }
    { B4 } { 017D }
    { B8 } { 017E }
    { BC } { 0152 }
    { BD } { 0153 }
    { BE } { 0178 }
  }
  {
  }
⟨/iso885915⟩
⟨*iso885916⟩
\__str_declare_eight_bit_encoding:nnnn { iso885916 } { 558 }
  {
    { A1 } { 0104 }
    { A2 } { 0105 }
    { A3 } { 0141 }
    { A4 } { 20AC }
    { A5 } { 201E }
    { A6 } { 0160 }
    { A8 } { 0161 }
    { AA } { 0218 }
    { AC } { 0179 }
    { AE } { 017A }
    { AF } { 017B }
    { B2 } { 010C }
    { B3 } { 0142 }
    { B4 } { 017D }
    { B5 } { 201D }
    { B8 } { 017E }
```

```
    { B9 } { 010D }
    { BA } { 0219 }
    { BC } { 0152 }
    { BD } { 0153 }
    { BE } { 0178 }
    { BF } { 017C }
    { C3 } { 0102 }
    { C5 } { 0106 }
    { D0 } { 0110 }
    { D1 } { 0143 }
    { D5 } { 0150 }
    { D7 } { 015A }
    { D8 } { 0170 }
    { DD } { 0118 }
    { DE } { 021A }
    { E3 } { 0103 }
    { E5 } { 0107 }
    { F0 } { 0111 }
    { F1 } { 0144 }
    { F5 } { 0151 }
    { F7 } { 015B }
    { F8 } { 0171 }
    { FD } { 0119 }
    { FE } { 021B }
  }
  {
  }
⟨/iso885916⟩
```

Chapter 57

l3quark implementation

The following test files are used for this code: m3quark001.lvt.

```
⟨*package⟩
```

57.1 Quarks

```
⟨@@=quark⟩
```

\quark_new:N

Allocate a new quark.

```
\cs_new_protected:Npn \quark_new:N #1
  {
    \__kernel_chk_if_free_cs:N #1
    \cs_gset_nopar:Npn #1 {#1}
  }
```

(*End of definition for* `\quark_new:N`. *This function is documented on page 148.*)

\q_nil
\q_mark
\q_no_value
\q_stop

Some "public" quarks. `\q_stop` is an "end of argument" marker, `\q_nil` is a empty value and `\q_no_value` marks an empty argument.

```
\quark_new:N \q_nil
\quark_new:N \q_mark
\quark_new:N \q_no_value
\quark_new:N \q_stop
```

(*End of definition for* `\q_nil` *and others. These variables are documented on page 148.*)

\q_recursion_tail
\q_recursion_stop

Quarks for ending recursions. Only ever used there! `\q_recursion_tail` is appended to whatever list structure we are doing recursion on, meaning it is added as a proper list item with whatever list separator is in use. `\q_recursion_stop` is placed directly after the list.

```
\quark_new:N \q_recursion_tail
\quark_new:N \q_recursion_stop
```

(*End of definition for* `\q_recursion_tail` *and* `\q_recursion_stop`. *These variables are documented on page 149.*)

\s__quark

Private scan mark used in l3quark. We don't have l3scan yet, so we declare the scan mark here and add it to the scan mark pool later.

```
\cs_new_eq:NN \s__quark \scan_stop:
```

(*End of definition for* `\s__quark`.)

`\q__quark_nil`

Private quark use for some tests.

```
\quark_new:N \q__quark_nil
```

(*End of definition for* `\q__quark_nil`.)

`\quark_if_recursion_tail_stop:N`
`\quark_if_recursion_tail_stop_do:Nn`

When doing recursions, it is easy to spend a lot of time testing if the end marker has been found. To avoid this, a dedicated end marker is used each time a recursion is set up. Thus if the marker is found everything can be wrapper up and finished off. The simple case is when the test can guarantee that only a single token is being tested. In this case, there is just a dedicated copy of the standard quark test. Both a gobbling version and one inserting end code are provided.

```
\cs_new:Npn \quark_if_recursion_tail_stop:N #1
  {
    \if_meaning:w \q_recursion_tail #1
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
  }
\cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
  {
    \if_meaning:w \q_recursion_tail #1
      \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
    \else:
      \exp_after:wN \use_none:n
    \fi:
  }
```

(*End of definition for* `\quark_if_recursion_tail_stop:N` *and* `\quark_if_recursion_tail_stop_do:Nn`. *These functions are documented on page* *149*.)

`\quark_if_recursion_tail_stop:n`
`\quark_if_recursion_tail_stop:o`
`\quark_if_recursion_tail_stop_do:nn`
`\quark_if_recursion_tail_stop_do:on`
`\__quark_if_recursion_tail:w`

See `\quark_if_nil:nTF` for the details. Expanding `\__quark_if_recursion_tail:w` once in front of the tokens chosen here gives an empty result if and only if `#1` is exactly `\q_recursion_tail`.

```
\cs_new:Npn \quark_if_recursion_tail_stop:n #1
  {
    \tl_if_empty:oTF
      { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
      { \use_none_delimit_by_q_recursion_stop:w }
      { }
  }
\cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
  {
    \tl_if_empty:oTF
      { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
      { \use_i_delimit_by_q_recursion_stop:nw }
      { \use_none:n }
  }
\cs_new:Npn \__quark_if_recursion_tail:w
    #1 \q_recursion_tail #2 ? #3 ?! { #1 #2 }
\cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
\cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn { o }
```

(*End of definition for* `\quark_if_recursion_tail_stop:n`, `\quark_if_recursion_tail_stop_do:nn`, *and* `\__quark_if_recursion_tail:w`. *These functions are documented on page* *149*.)

\quark_if_recursion_tail_break:NN
\quark_if_recursion_tail_break:nN

Analogues of the \quark_if_recursion_tail_stop... functions. Break the mapping using #2.

```
\cs_new:Npn \quark_if_recursion_tail_break:NN #1#2
  {
    \if_meaning:w \q_recursion_tail #1
      \exp_after:wN #2
    \fi:
  }
\cs_new:Npn \quark_if_recursion_tail_break:nN #1#2
  {
    \tl_if_empty:oT
      { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
      {#2}
  }
```

(*End of definition for* \quark_if_recursion_tail_break:NN *and* \quark_if_recursion_tail_break:nN*. These functions are documented on page 150.*)

\quark_if_nil_p:N
\quark_if_nil:NTF
\quark_if_no_value_p:N
\quark_if_no_value_p:c
\quark_if_no_value:NTF
\quark_if_no_value:cTF

Here we test if we found a special quark as the first argument. We better start with \q_no_value as the first argument since the whole thing may otherwise loop if #1 is wrongly given a string like aabc instead of a single token.[9]

```
\prg_new_conditional:Npnn \quark_if_nil:N #1 { p, T , F , TF }
  {
    \if_meaning:w \q_nil #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_new_conditional:Npnn \quark_if_no_value:N #1 { p, T , F , TF }
  {
    \if_meaning:w \q_no_value #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_generate_conditional_variant:Nnn \quark_if_no_value:N
  { c } { p , T , F , TF }
```

(*End of definition for* \quark_if_nil:NTF *and* \quark_if_no_value:NTF*. These functions are documented on page 148.*)

\quark_if_nil_p:n
\quark_if_nil_p:V
\quark_if_nil_p:o
\quark_if_nil:nTF
\quark_if_nil:VTF
\quark_if_nil:oTF
\quark_if_no_value_p:n
\quark_if_no_value:nTF
__quark_if_nil:w
__quark_if_no_value:w
__quark_if_empty_if:o

Let us explain \quark_if_nil:nTF. Expanding __quark_if_nil:w once is safe thanks to the trailing \q_nil ??!. The result of expanding once is empty if and only if both delimited arguments #1 and #2 are empty and #3 is delimited by the last tokens ?!. Thanks to the leading {}, the argument #1 is empty if and only if the argument of \quark_if_nil:n starts with \q_nil. The argument #2 is empty if and only if this \q_nil is followed immediately by ? or by {}?, coming either from the trailing tokens in the definition of \quark_if_nil:n, or from its argument. In the first case, __quark_if_nil:w is followed by {}\q_nil {}? !\q_nil ??!, hence #3 is delimited by the final ?!, and the test returns true as wanted. In the second case, the result is not empty since

[9] It may still loop in special circumstances however!

the first ?! in the definition of \quark_if_nil:n stop #3. The auxiliary here is the same as __tl_if_empty_if:o, with the same comments applying.

```
\prg_new_conditional:Npnn \quark_if_nil:n #1 { p, T , F , TF }
  {
    \__quark_if_empty_if:o
      { \__quark_if_nil:w {} #1 {} ? ! \q_nil ? ? ! }
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \__quark_if_nil:w #1 \q_nil #2 ? #3 ? ! { #1 #2 }
\prg_new_conditional:Npnn \quark_if_no_value:n #1 { p, T , F , TF }
  {
    \__quark_if_empty_if:o
      { \__quark_if_no_value:w {} #1 {} ? ! \q_no_value ? ? ! }
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \__quark_if_no_value:w #1 \q_no_value #2 ? #3 ? ! { #1 #2 }
\prg_generate_conditional_variant:Nnn \quark_if_nil:n
  { V , o } { p , TF , T , F }
\cs_new:Npn \__quark_if_empty_if:o #1
  {
    \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
      \__kernel_tl_to_str:w \exp_after:wN {#1} \q_nil
  }
```

(*End of definition for* \quark_if_nil:nTF *and others. These functions are documented on page 148.*)

__kernel_quark_new_test:N

The function __kernel_quark_new_test:N defines #1 in a similar way as \quark_if_recursion_tail_... functions (as described below), using \q__⟨*namespace*⟩_recursion_tail as the test quark and \q__⟨*namespace*⟩_recursion_stop as the delimiter quark, where the ⟨*namespace*⟩ is determined as the first _-delimited part in #1.

There are six possible function types which this function can define, and which is defined depends on the signature of the function being defined:

:n gives an analogue of \quark_if_recursion_tail_stop:n

:nn gives an analogue of \quark_if_recursion_tail_stop_do:nn

:nN gives an analogue of \quark_if_recursion_tail_break:nN

:N gives an analogue of \quark_if_recursion_tail_stop:N

:Nn gives an analogue of \quark_if_recursion_tail_stop_do:Nn

:NN gives an analogue of \quark_if_recursion_tail_break:NN

Any other signature causes an error, as does a function without signature.

__kernel_quark_new_conditional:Nn

Similar to `\__kernel_quark_new_test:N`, but defines quark branching conditionals like `\quark_if_nil:nTF` that test for the quark `\q__`⟨*namespace*⟩`_`⟨*name*⟩. The ⟨*namespace*⟩ and ⟨*name*⟩ are determined from the conditional `#1`, which must take the rather rigid form `\__`⟨*namespace*⟩`_quark_if_`⟨*name*⟩`:`⟨*arg spec*⟩. There are only two cases for the ⟨*arg spec*⟩ here:

`:n` gives an analogue of `\quark_if_nil:nTF`

`:N` gives an analogue of `\quark_if_nil:NTF`

Any other signature causes an error, as does a function without signature. We use low-level emptiness tests as l3tl is not available yet when these functions are used; thankfully we only care about whether strings are empty so a simple `\if_meaning:w \q_nil` ⟨*string*⟩ `\q_nil` suffices.

__quark_new_test:NNNn
__quark_new_test:Nccn
__quark_new_test_aux:nnNNnnnn
__quark_new_conditional:Nnnn
__quark_new_conditional:Neen

```
\cs_new_protected:Npn \__kernel_quark_new_test:N #1
  { \__quark_new_test_aux:Ne #1 { \__quark_module_name:N #1 } }
\cs_new_protected:Npn \__quark_new_test_aux:Nn #1 #2
  {
    \if_meaning:w \q_nil #2 \q_nil
      \msg_error:nne { quark } { invalid-function }
        { \token_to_str:N #1 }
    \else:
      \__quark_new_test:Nccn #1
        { q__#2_recursion_tail } { q__#2_recursion_stop } { __#2 }
    \fi:
  }
\cs_generate_variant:Nn \__quark_new_test_aux:Nn { Ne }
\cs_new_protected:Npn \__quark_new_test:NNNn #1
  {
    \exp_last_unbraced:Nf \__quark_new_test_aux:nnNNnnnn
      { \cs_split_function:N #1 }
      #1 { test }
  }
\cs_generate_variant:Nn \__quark_new_test:NNNn { Ncc }
\cs_new_protected:Npn \__kernel_quark_new_conditional:Nn #1
  {
    \__quark_new_conditional:Neen #1
      { \__quark_quark_conditional_name:N #1 }
      { \__quark_module_name:N #1 }
  }
\cs_new_protected:Npn \__quark_new_conditional:Nnnn #1#2#3#4
  {
    \if_meaning:w \q_nil #2 \q_nil
      \msg_error:nne { quark } { invalid-function }
        { \token_to_str:N #1 }
    \else:
      \if_meaning:w \q_nil #3 \q_nil
        \msg_error:nne { quark } { invalid-function }
          { \token_to_str:N #1 }
      \else:
        \exp_last_unbraced:Nf \__quark_new_test_aux:nnNNnnnn
          { \cs_split_function:N #1 }
```

```
          #1 { conditional }
          {#2} {#3} {#4}
      \fi:
    \fi:
  }
\cs_generate_variant:Nn \__quark_new_conditional:Nnnn { Nee }
\cs_new_protected:Npn \__quark_new_test_aux:nnNNnnnn #1 #2 #3 #4 #5
  {
    \cs_if_exist_use:cTF { __quark_new_#5_#2:Nnnn } { #4 }
      {
        \msg_error:nnee { quark } { invalid-function }
          { \token_to_str:N #4 } {#2}
        \use_none:nnn
      }
  }
```

(*End of definition for* `\__kernel_quark_new_test:N` *and others.*)

`\__quark_new_test_n:Nnnn`
`\__quark_new_test_nn:Nnnn`
`\__quark_new_test_N:Nnnn`
`\__quark_new_test_Nn:Nnnn`
`\__quark_new_test_NN:Nnnn`
`\__quark_new_test_NN:Nnnn`

These macros implement the six possibilities mentioned above, passing the right arguments to `\__quark_new_test_aux_do:nNNnnnnNNn`, which defines some auxiliaries, and then to `\__quark_new_test_define_tl:nNnNNn` (`:n(n)` variants) or to `\__quark_new_-test_define_ifx:nNnNNn` (`:N(n)`) which define the main conditionals.

```
\cs_new_protected:Npn \__quark_new_test_n:Nnnn #1 #2 #3 #4
  {
    \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { none } { } { } { }
      \__quark_new_test_define_tl:nNnNNn #1 { }
  }
\cs_new_protected:Npn \__quark_new_test_nn:Nnnn #1 #2 #3 #4
  {
    \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
      \__quark_new_test_define_tl:nNnNNn #1 { \use_none:n }
  }
\cs_new_protected:Npn \__quark_new_test_nN:Nnnn #1 #2 #3 #4
  {
    \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
      \__quark_new_test_define_break_tl:nNNNNn #1 { }
  }
\cs_new_protected:Npn \__quark_new_test_N:Nnnn #1 #2 #3 #4
  {
    \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { none } { } { } { }
      \__quark_new_test_define_ifx:nNnNNn #1 { }
  }
\cs_new_protected:Npn \__quark_new_test_Nn:Nnnn #1 #2 #3 #4
  {
    \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
      \__quark_new_test_define_ifx:nNnNNn #1
      { \else: \exp_after:wN \use_none:n }
  }
\cs_new_protected:Npn \__quark_new_test_NN:Nnnn #1 #2 #3 #4
  {
    \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
      \__quark_new_test_define_break_ifx:nNNNNn #1 { }
  }
```

(*End of definition for* `\__quark_new_test_n:Nnnn` *and others.*)

__quark_new_test_aux_do:nNNnnnnNNn
__quark_test_define_aux:NNNNnnNNn

`\__quark_new_test_aux_do:nNNnnnnNNn` makes the control sequence names which will be used by `\__quark_test_define_aux:NNNNnnNNn`, and then later by `\__quark_new_test_define_tl:nNnNNn` or `\__quark_new_test_define_ifx:nNnNNn`. The control sequences defined here are analogous to `\__quark_if_recursion_tail:w` and to `\use_(none|i)_delimit_by_q_recursion_stop:(|n)w`.

The name is composed by the name-space and the name of the quarks. Suppose `\__kernel_quark_new_test:N` was used with:

```
\__kernel_quark_new_test:N \__test_quark_tail:n
```

then the first auxiliary will be `\__test_quark_recursion_tail:w`, and the second one will be `\__test_use_none_delimit_by_q_recursion_stop:w`.

Note that the actual quarks are *not* defined here. They should be defined separately using `\quark_new:N`.

```
\cs_new_protected:Npn \__quark_new_test_aux_do:nNNnnnnNNn #1 #2 #3 #4 #5
  {
    \exp_args:Ncc \__quark_test_define_aux:NNNNnnNNn
      { #1 _quark_recursion_tail:w }
      { #1 _use_ #4 _delimit_by_q_recursion_stop: #5 w }
      #2 #3
  }
\cs_new_protected:Npn \__quark_test_define_aux:NNNNnnNNn #1 #2 #3 #4 #5 #6 #7
  {
    \cs_gset:Npn #1  ##1 #3 ##2 ? ##3 ?! { ##1 ##2 }
    \cs_gset:Npn #2  ##1 #6 #4 {#5}
    #7 {##1} #1 #2 #3
  }
```

(*End of definition for* `\__quark_new_test_aux_do:nNNnnnnNNn` *and* `\__quark_test_define_aux:NNNNnnNNn`.)

__quark_new_test_define_tl:nNnNNn
__quark_new_test_define_ifx:nNnNNn
__quark_new_test_define_break_tl:nNNNNn
__quark_new_test_define_break_ifx:nNNNNn

Finally, these two macros define the main conditional function using what's been set up before.

```
\cs_new_protected:Npn \__quark_new_test_define_tl:nNnNNn #1 #2 #3 #4 #5 #6
  {
    \cs_new:Npn #5 #1
      {
        \tl_if_empty:oTF
          { #2 {} ##1 {} ?! #4 ??! }
          {#3} {#6}
      }
  }
\cs_new_protected:Npn \__quark_new_test_define_ifx:nNnNNn #1 #2 #3 #4 #5 #6
  {
    \cs_new:Npn #5 #1
      {
        \if_meaning:w #4 ##1
          \exp_after:wN #3
          #6
        \fi:
      }
  }
\cs_new_protected:Npn \__quark_new_test_define_break_tl:nNNNNn #1 #2 #3
  { \__quark_new_test_define_tl:nNnNNn {##1##2} #2 {##2} }
\cs_new_protected:Npn \__quark_new_test_define_break_ifx:nNNNNn #1 #2 #3
```

```
  { \__quark_new_test_define_ifx:nNnNNn {##1##2} #2 {##2} }
```

(*End of definition for* `\__quark_new_test_define_tl:nNnNNn` *and others.*)

`\__quark_new_conditional_n:Nnnn`
`\__quark_new_conditional_N:Nnnn`
`\__quark_new_conditional_n_aux:NNNn`
`\__quark_new_conditional_N_aux:NNNn`

These macros implement the two possibilities for branching quark conditionals. To avoid constructing without defining the `\__`⟨*type*⟩`_if_quark_`⟨*name*⟩`:w` helper, `N`-type function accepts a `\prg_do_nothing:` as a placeholder.

```
\cs_new_protected:Npn \__quark_new_conditional_n:Nnnn #1 #2 #3
  {
    \exp_args:Ncc \__quark_new_conditional_n_aux:NNNn
      { __ #3 _if_quark_ #2 :w } { q__ #3 _ #2 } #1
  }
\cs_new_protected:Npn \__quark_new_conditional_N:Nnnn #1 #2 #3
  {
    \exp_args:NNc \__quark_new_conditional_N_aux:NNNn
      \prg_do_nothing: { q__ #3 _ #2 } #1
  }
\cs_new_protected:Npn \__quark_new_conditional_n_aux:NNNn #1 #2 #3 #4
  {
    \cs_gset:Npn #1 ##1 #2 ##2 ? ##3 ?! { ##1##2 }
    \prg_new_conditional:Npnn #3 ##1 {#4}
      {
        \__quark_if_empty_if:o { #1 {} ##1 {} ?! #2 ??! }
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      }
  }
\cs_new_protected:Npn \__quark_new_conditional_N_aux:NNNn #1 #2 #3 #4
  {
    \prg_new_conditional:Npnn #3 ##1 {#4}
      {
        \if_meaning:w #2 ##1
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      }
  }
```

(*End of definition for* `\__quark_new_conditional_n:Nnnn` *and others.*)

`\__quark_module_name:N`
`\__quark_module_name:w`
`\__quark_module_name_loop:w`
`\__quark_module_name_end:w`

`\__quark_module_name:N` takes a control sequence and returns its ⟨*module*⟩ name, determined as the first non-empty non-single-character word, separated by `_` or `:`. These rules give the correct result for public functions `\`⟨*module*⟩`_...`, private functions `\__`⟨*module*⟩`_...`, and variables such as `\l_`⟨*module*⟩`_...`. If no valid module is found the result is an empty string. The approach is to first cut off everything after the (first) `:` if any is present, then repeatedly grab `_`-delimited words until finding one of length at least 2 (we use low-level tests as l3tl is not fully available when `\__kernel_quark_new_test:N` is first used. If no ⟨*module*⟩ is found (such as in `\::n`) we get the trailing marker `\use_none:n {}`, which expands to nothing.

```
\cs_set:Npn \__quark_tmp:w #1#2
  {
```

```
    \cs_new:Npn \__quark_module_name:N ##1
      {
        \exp_last_unbraced:Nf \__quark_module_name:w
          { \cs_to_str:N ##1 } #1 \s__quark
      }
    \cs_new:Npn \__quark_module_name:w ##1 #1 ##2 \s__quark
      { \__quark_module_name_loop:w ##1 #2 \use_none:n { } #2 \s__quark }
    \cs_new:Npn \__quark_module_name_loop:w ##1 #2
      {
        \use_i_ii:nnn \if_meaning:w \prg_do_nothing:
            ##1 \prg_do_nothing: \prg_do_nothing:
          \exp_after:wN \__quark_module_name_loop:w
        \else:
          \__quark_module_name_end:w ##1
        \fi:
      }
    \cs_new:Npn \__quark_module_name_end:w
      ##1 \fi: ##2 \s__quark { \fi: ##1 }
  }
\exp_after:wN \__quark_tmp:w \tl_to_str:n { : _ }
```

(*End of definition for* `\__quark_module_name:N` *and others.*)

__quark_quark_conditional_name:N
__quark_quark_conditional_name:w

`\__quark_quark_conditional_name:N` determines the quark name that the quark conditional function `##1` queries, as the part of the function name between `_quark_if_` and the trailing `:`. Again we define it through `\__quark_tmp:w`, which receives `:` as `#1` and `_quark_if_` as `#2`. The auxiliary `\__quark_quark_conditional_name:w` returns the part between the first `_quark_if_` and the next `:`, and we apply this auxiliary to the function name followed by `:` (in case the function name is lacking a signature), and `_quark_if_:` so that `\__quark_quark_conditional_name:N` returns an empty string if `_quark_if_` is not present.

```
\cs_set:Npn \__quark_tmp:w #1 #2 \s__quark
  {
    \cs_new:Npn \__quark_quark_conditional_name:N ##1
      {
        \exp_last_unbraced:Nf \__quark_quark_conditional_name:w
          { \cs_to_str:N ##1 } #1 #2 #1 \s__quark
      }
    \cs_new:Npn \__quark_quark_conditional_name:w
      ##1 #2 ##2 #1 ##3 \s__quark {##2}
  }
\exp_after:wN \__quark_tmp:w \tl_to_str:n { : _quark_if_ } \s__quark
```

(*End of definition for* `\__quark_quark_conditional_name:N` *and* `\__quark_quark_conditional_name:w`*.*)

57.2 Scan marks

```
⟨@@=scan⟩
```

\scan_new:N

Check whether the variable is already a scan mark, then declare it to be equal to `\scan_stop:` globally.

```
\cs_new_protected:Npn \scan_new:N #1
  {
```

```
    \tl_if_in:NnTF \g__scan_marks_tl { #1 }
      {
        \msg_error:nne { scanmark } { already-defined }
          { \token_to_str:N #1 }
      }
      {
        \tl_gput_right:Nn \g__scan_marks_tl {#1}
        \cs_new_eq:NN #1 \scan_stop:
      }
  }
```

(*End of definition for* `\scan_new:N`. *This function is documented on page 151.*)

`\s_stop`
`\g__scan_marks_tl`

We only declare one scan mark here, more can be defined by specific modules. Can't use `\scan_new:N` yet because l3tl isn't loaded, so define `\s_stop` by hand and add it to `\g__scan_marks_tl`. We also add the scan marks declared earlier to the pool here. Since they lives in a different namespace, a little DocStrip cheating is necessary.

```
\cs_new_eq:NN \s_stop \scan_stop:
\cs_gset_nopar:Npn \g__scan_marks_tl
  {
    \s_stop
⟨@@=quark⟩
    \s__quark
⟨@@=cs⟩
    \s__cs_mark
    \s__cs_stop
⟨@@=scan⟩
  }
```

(*End of definition for* `\s_stop` *and* `\g__scan_marks_tl`. *This variable is documented on page 151.*)

`\use_none_delimit_by_s_stop:w`

Similar to `\use_none_delimit_by_q_stop:w`.

```
\cs_new:Npn \use_none_delimit_by_s_stop:w #1 \s_stop { }
```

(*End of definition for* `\use_none_delimit_by_s_stop:w`. *This function is documented on page 151.*)

```
⟨/package⟩
```

Chapter 58

l3seq implementation

The following test files are used for this code: m3seq002,m3seq003.

```
⟨*package⟩
⟨@@=seq⟩
```

A sequence is a control sequence whose top-level expansion is of the form "`\s__seq \__seq_item:n {⟨item_1⟩} ... \__seq_item:n {⟨item_n⟩}`", with a leading scan mark followed by n items of the same form. An earlier implementation used the structure "`\seq_elt:w ⟨item_1⟩ \seq_elt_end: ... \seq_elt:w ⟨item_n⟩ \seq_elt_end:`". This allowed rapid searching using a delimited function, but was not suitable for items containing `{`, `}` and `#` tokens, and also lead to the loss of surrounding braces around items

`\__seq_item:n` ⋆

`\__seq_item:n {⟨item⟩}`

The internal token used to begin each sequence entry. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.

`\__seq_push_item_def:n`
`\__seq_push_item_def:e`

`\__seq_push_item_def:n {⟨code⟩}`

Saves the definition of `\__seq_item:n` and redefines it to accept one parameter and expand to ⟨*code*⟩. This function should always be balanced by use of `\__seq_pop_item_def:`.

`\__seq_pop_item_def:`

`\__seq_pop_item_def:`

Restores the definition of `\__seq_item:n` most recently saved by `\__seq_push_item_def:n`. This function should always be used in a balanced pair with `\__seq_push_item_def:n`.

`\s__seq` This private scan mark.

```
\scan_new:N \s__seq
```

(*End of definition for* `\s__seq`.)

`\s__seq_mark`
`\s__seq_stop`

Private scan marks.

```
\scan_new:N \s__seq_mark
\scan_new:N \s__seq_stop
```

(*End of definition for* `\s__seq_mark` *and* `\s__seq_stop`.)

__seq_item:n

The delimiter is always defined, but when used incorrectly simply removes its argument and hits an undefined control sequence to raise an error.

```
\cs_new:Npn \__seq_item:n
  {
    \msg_expandable_error:nn { seq } { misused }
    \use_none:n
  }
```

(*End of definition for* `\__seq_item:n`.)

\l__seq_internal_a_tl
\l__seq_internal_b_tl

Scratch space for various internal uses.

```
\tl_new:N \l__seq_internal_a_tl
\tl_new:N \l__seq_internal_b_tl
```

(*End of definition for* `\l__seq_internal_a_tl` *and* `\l__seq_internal_b_tl`.)

__seq_tmp:w

Scratch function for internal use.

```
\cs_new_eq:NN \__seq_tmp:w ?
```

(*End of definition for* `\__seq_tmp:w`.)

\c_empty_seq

A sequence with no item, following the structure mentioned above.

```
\tl_const:Nn \c_empty_seq { \s__seq }
```

(*End of definition for* `\c_empty_seq`. *This variable is documented on page 164.*)

58.1 Allocation and initialisation

\seq_new:N
\seq_new:c

Sequences are initialized to `\c_empty_seq`.

```
\cs_new_protected:Npn \seq_new:N #1
  {
    \__kernel_chk_if_free_cs:N #1
    \cs_gset_eq:NN #1 \c_empty_seq
  }
\cs_generate_variant:Nn \seq_new:N { c }
```

(*End of definition for* `\seq_new:N`. *This function is documented on page 152.*)

\seq_clear:N
\seq_clear:c
\seq_gclear:N
\seq_gclear:c

Clearing a sequence is similar to setting it equal to the empty one.

```
\cs_new_protected:Npn \seq_clear:N  #1
  { \seq_set_eq:NN #1 \c_empty_seq }
\cs_generate_variant:Nn \seq_clear:N  { c }
\cs_new_protected:Npn \seq_gclear:N #1
  { \seq_gset_eq:NN #1 \c_empty_seq }
\cs_generate_variant:Nn \seq_gclear:N { c }
```

(*End of definition for* `\seq_clear:N` *and* `\seq_gclear:N`. *These functions are documented on page 152.*)

\seq_clear_new:N
\seq_clear_new:c
\seq_gclear_new:N
\seq_gclear_new:c

Once again we copy code from the token list functions.

```
\cs_new_protected:Npn \seq_clear_new:N  #1
  { \seq_if_exist:NTF #1 { \seq_clear:N #1 } { \seq_new:N #1 } }
\cs_generate_variant:Nn \seq_clear_new:N  { c }
\cs_new_protected:Npn \seq_gclear_new:N #1
  { \seq_if_exist:NTF #1 { \seq_gclear:N #1 } { \seq_new:N #1 } }
\cs_generate_variant:Nn \seq_gclear_new:N { c }
```

(*End of definition for* \seq_clear_new:N *and* \seq_gclear_new:N. *These functions are documented on page 152.*)

\seq_set_eq:NN
\seq_set_eq:cN
\seq_set_eq:Nc
\seq_set_eq:cc
\seq_gset_eq:NN
\seq_gset_eq:cN
\seq_gset_eq:Nc
\seq_gset_eq:cc

Copying a sequence is the same as copying the underlying token list.

```
\cs_new_eq:NN \seq_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \seq_set_eq:Nc  \tl_set_eq:Nc
\cs_new_eq:NN \seq_set_eq:cN  \tl_set_eq:cN
\cs_new_eq:NN \seq_set_eq:cc  \tl_set_eq:cc
\cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
\cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
\cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN
\cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc
```

(*End of definition for* \seq_set_eq:NN *and* \seq_gset_eq:NN. *These functions are documented on page 152.*)

\seq_set_from_clist:NN
\seq_set_from_clist:cN
\seq_set_from_clist:Nc
\seq_set_from_clist:cc
\seq_set_from_clist:Nn
\seq_set_from_clist:cn
\seq_gset_from_clist:NN
\seq_gset_from_clist:cN
\seq_gset_from_clist:Nc
\seq_gset_from_clist:cc
\seq_gset_from_clist:Nn
\seq_gset_from_clist:cn

Setting a sequence from a comma-separated list is done using a simple mapping.

```
\cs_new_protected:Npn \seq_set_from_clist:NN #1#2
  {
    \__kernel_tl_set:Nx #1
      { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
  }
\cs_new_protected:Npn \seq_set_from_clist:Nn #1#2
  {
    \__kernel_tl_set:Nx #1
      { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
  }
\cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
  {
    \__kernel_tl_gset:Nx #1
      { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
  }
\cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
  {
    \__kernel_tl_gset:Nx #1
      { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
  }
\cs_generate_variant:Nn \seq_set_from_clist:NN  {     Nc }
\cs_generate_variant:Nn \seq_set_from_clist:NN  { c , cc }
\cs_generate_variant:Nn \seq_set_from_clist:Nn  { c      }
\cs_generate_variant:Nn \seq_gset_from_clist:NN {     Nc }
\cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
\cs_generate_variant:Nn \seq_gset_from_clist:Nn { c      }
```

(*End of definition for* \seq_set_from_clist:NN *and others. These functions are documented on page 153.*)

\seq_const_from_clist:Nn
\seq_const_from_clist:cn

Almost identical to \seq_set_from_clist:Nn.

```
\cs_new_protected:Npn \seq_const_from_clist:Nn #1#2
  {
    \tl_const:Ne #1
      { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
  }
\cs_generate_variant:Nn \seq_const_from_clist:Nn { c }
```

(*End of definition for* `\seq_const_from_clist:Nn`. *This function is documented on page* 153.)

`\seq_set_split:Nnn`
`\seq_set_split:NVn`
`\seq_set_split:NnV`
`\seq_set_split:NVV`
`\seq_set_split:Nne`
`\seq_set_split:Nee`
`\seq_set_split:Nnx`
`\seq_set_split:Nxx`
`\seq_gset_split:Nnn`
`\seq_gset_split:NVn`
`\seq_gset_split:NnV`
`\seq_gset_split:NVV`
`\seq_gset_split:Nne`
`\seq_gset_split:Nee`
`\seq_gset_split:Nnx`
`\seq_gset_split:Nxx`
`\seq_set_split_keep_spaces:Nnn`
`\seq_set_split_keep_spaces:NnV`
`\seq_gset_split_keep_spaces:Nnn`
`\seq_gset_split_keep_spaces:NnV`
`\__seq_set_split:NNNnn`
`\__seq_set_split:Nw`
`\__seq_set_split:w`
`\__seq_set_split_end:`

When the separator is empty, everything is very simple, just map `\__seq_wrap_item:n` through the items of the last argument. For non-trivial separators, the goal is to split a given token list at the marker, strip spaces from each item, and remove one set of outer braces if after removing leading and trailing spaces the item is enclosed within braces. After `\tl_replace_all:Nnn`, the token list `\l__seq_internal_a_tl` is a repetition of the pattern `\__seq_set_split:Nw \prg_do_nothing:` ⟨*item with spaces*⟩ `\__seq_set_split_end:`. Then, e-expansion causes `\__seq_set_split:Nw` to trim spaces, and leaves its result as `\__seq_set_split:w` ⟨*trimmed item*⟩ `\__seq_set_split_end:`. This is then converted to the l3seq internal structure by another e-expansion. In the first step, we insert `\prg_do_nothing:` to avoid losing braces too early: that would cause space trimming to act within those lost braces. The second step is solely there to strip braces which are outermost after space trimming.

```
\cs_new_protected:Npn \seq_set_split:Nnn
  { \__seq_set_split:NNNnn \__kernel_tl_set:Nx \tl_trim_spaces:n }
\cs_new_protected:Npn \seq_gset_split:Nnn
  { \__seq_set_split:NNNnn \__kernel_tl_gset:Nx \tl_trim_spaces:n }
\cs_new_protected:Npn \seq_set_split_keep_spaces:Nnn
  { \__seq_set_split:NNNnn \__kernel_tl_set:Nx \exp_not:n }
\cs_new_protected:Npn \seq_gset_split_keep_spaces:Nnn
  { \__seq_set_split:NNNnn \__kernel_tl_gset:Nx \exp_not:n }
\cs_new_protected:Npn \__seq_set_split:NNNnn #1#2#3#4#5
  {
    \tl_if_empty:nTF {#4}
      {
        \tl_set:Nn \l__seq_internal_a_tl
          { \tl_map_function:nN {#5} \__seq_wrap_item:n }
      }
      {
        \tl_set:Nn \l__seq_internal_a_tl
          {
            \__seq_set_split:Nw #2 \prg_do_nothing:
            #5
            \__seq_set_split_end:
          }
        \tl_replace_all:Nnn \l__seq_internal_a_tl {#4}
          {
            \__seq_set_split_end:
            \__seq_set_split:Nw #2 \prg_do_nothing:
          }
        \__kernel_tl_set:Nx \l__seq_internal_a_tl { \l__seq_internal_a_tl }
      }
    #1 #3 { \s__seq \l__seq_internal_a_tl }
  }
\cs_new:Npn \__seq_set_split:Nw #1#2 \__seq_set_split_end:
  {
    \exp_not:N \__seq_set_split:w
    \exp_args:No #1 {#2}
    \exp_not:N \__seq_set_split_end:
  }
\cs_new:Npn \__seq_set_split:w #1 \__seq_set_split_end:
  { \__seq_wrap_item:n {#1} }
```

```
\cs_generate_variant:Nn \seq_set_split:Nnn  { NV , NnV , NVV , Nne , Nee }
\cs_generate_variant:Nn \seq_set_split:Nnn  { Nnx , Nxx }
\cs_generate_variant:Nn \seq_gset_split:Nnn { NV , NnV , NVV , Nne , Nee }
\cs_generate_variant:Nn \seq_gset_split:Nnn { Nnx , Nxx }
\cs_generate_variant:Nn \seq_set_split_keep_spaces:Nnn  { NnV }
\cs_generate_variant:Nn \seq_gset_split_keep_spaces:Nnn { NnV }
```

(*End of definition for* `\seq_set_split:Nnn` *and others. These functions are documented on page* *153*.)

`\seq_set_filter:NNn`
`\seq_gset_filter:NNn`
`\__seq_set_filter:NNNn`

Similar to `\seq_map_inline:Nn`, without a `\prg_break_point:` because the user's code is performed within the evaluation of a boolean expression, and skipping out of that would break horribly. The `\__seq_wrap_item:n` function inserts the relevant `\__seq_item:n` without expansion in the input stream, hence in the `e`-expanding assignment.

```
\cs_new_protected:Npn \seq_set_filter:NNn
  { \__seq_set_filter:NNNn \__kernel_tl_set:Nx }
\cs_new_protected:Npn \seq_gset_filter:NNn
  { \__seq_set_filter:NNNn \__kernel_tl_gset:Nx }
\cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
  {
    \__seq_push_item_def:n { \bool_if:nT {#4} { \__seq_wrap_item:n {##1} } }
    #1 #2 { #3 }
    \__seq_pop_item_def:
  }
```

(*End of definition for* `\seq_set_filter:NNn`*,* `\seq_gset_filter:NNn`*, and* `\__seq_set_filter:NNNn`*. These functions are documented on page* *154*.)

`\seq_concat:NNN`
`\seq_concat:ccc`
`\seq_gconcat:NNN`
`\seq_gconcat:ccc`

When concatenating sequences, one must remove the leading `\s__seq` of the second sequence. The result starts with `\s__seq` (of the first sequence), which stops `f`-expansion.

```
\cs_new_protected:Npn \seq_concat:NNN #1#2#3
  { \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
\cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
  { \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
\cs_generate_variant:Nn \seq_concat:NNN  { ccc }
\cs_generate_variant:Nn \seq_gconcat:NNN { ccc }
```

(*End of definition for* `\seq_concat:NNN` *and* `\seq_gconcat:NNN`*. These functions are documented on page* *154*.)

`\seq_if_exist_p:N`
`\seq_if_exist_p:c`
`\seq_if_exist:NTF`
`\seq_if_exist:cTF`

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \seq_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \seq_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End of definition for* `\seq_if_exist:NTF`*. This function is documented on page* *154*.)

58.2 Appending data to either end

`\seq_put_left:Nn`
`\seq_put_left:NV`
`\seq_put_left:Nv`
`\seq_put_left:Ne`
`\seq_put_left:No`
`\seq_put_left:Nx`
`\seq_put_left:cn`
`\seq_put_left:cV`
`\seq_put_left:cv`
`\seq_put_left:ce`
`\seq_put_left:co`
`\seq_put_left:cx`
`\seq_gput_left:Nn`
`\seq_gput_left:NV`
`\seq_gput_left:Nv`

When adding to the left of a sequence, remove `\s__seq`. This is done by `\__seq_put_left_aux:w`, which also stops `f`-expansion.

```
\cs_new_protected:Npn \seq_put_left:Nn #1#2
  {
    \__kernel_tl_set:Nx #1
```

```
      {
        \exp_not:n { \s__seq \__seq_item:n {#2} }
        \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
      }
  }
\cs_new_protected:Npn \seq_gput_left:Nn #1#2
  {
    \__kernel_tl_gset:Nx #1
      {
        \exp_not:n { \s__seq \__seq_item:n {#2} }
        \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
      }
  }
\cs_new:Npn \__seq_put_left_aux:w \s__seq { \exp_stop_f: }
\cs_generate_variant:Nn \seq_put_left:Nn  {     NV , Nv , Ne , No , Nx }
\cs_generate_variant:Nn \seq_put_left:Nn  { c , cV , cv , ce , co ,cx }
\cs_generate_variant:Nn \seq_gput_left:Nn {     NV , Nv , Ne , No , Nx }
\cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , ce , co , cx }
```

(*End of definition for* `\seq_put_left:Nn`*,* `\seq_gput_left:Nn`*, and* `\__seq_put_left_aux:w`*. These functions are documented on page* 154*.*)

`\seq_put_right:Nn` `\seq_put_right:NV` `\seq_put_right:Nv` `\seq_put_right:Ne` `\seq_put_right:No` `\seq_put_right:Nx` `\seq_put_right:cn` `\seq_put_right:cV` `\seq_put_right:cv` `\seq_put_right:cx` `\seq_put_right:co` `\seq_put_right:cx` `\seq_gput_right:Nn` `\seq_gput_right:NV` `\seq_gput_right:Nv` `\seq_gput_right:Ne` `\seq_gput_right:No` `\seq_gput_right:Nx` `\seq_gput_right:cn` `\seq_gput_right:cV` `\seq_gput_right:cv` `\seq_gput_right:ce` `\seq_gput_right:co` `\seq_gput_right:cx`

Since there is no trailing marker, adding an item to the right of a sequence simply means wrapping it in `\__seq_item:n`.

```
\cs_new_protected:Npn \seq_put_right:Nn #1#2
  { \tl_put_right:Nn #1 { \__seq_item:n {#2} } }
\cs_new_protected:Npn \seq_gput_right:Nn #1#2
  { \tl_gput_right:Nn #1 { \__seq_item:n {#2} } }
\cs_generate_variant:Nn \seq_put_right:Nn  {     NV , Nv , Ne , No , Nx }
\cs_generate_variant:Nn \seq_put_right:Nn  { c , cV , cv , ce , co , cx }
\cs_generate_variant:Nn \seq_gput_right:Nn {     NV , Nv , Ne , No , Nx }
\cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , ce , co , cx }
```

(*End of definition for* `\seq_put_right:Nn` *and* `\seq_gput_right:Nn`*. These functions are documented on page* 154*.*)

58.3 Modifying sequences

`\__seq_wrap_item:n`

This function converts its argument to a proper sequence item in an e-expansion context.

```
\cs_new:Npn \__seq_wrap_item:n #1 { \exp_not:n { \__seq_item:n {#1} } }
```

(*End of definition for* `\__seq_wrap_item:n`*.*)

`\l__seq_remove_seq`

An internal sequence for the removal routines.

```
\seq_new:N \l__seq_remove_seq
```

(*End of definition for* `\l__seq_remove_seq`*.*)

`\seq_remove_duplicates:N` `\seq_remove_duplicates:c` `\seq_gremove_duplicates:N` `\seq_gremove_duplicates:c` `\__seq_remove_duplicates:NN`

Removing duplicates means making a new list then copying it.

```
\cs_new_protected:Npn \seq_remove_duplicates:N
  { \__seq_remove_duplicates:NN \seq_set_eq:NN }
\cs_new_protected:Npn \seq_gremove_duplicates:N
  { \__seq_remove_duplicates:NN \seq_gset_eq:NN }
\cs_new_protected:Npn \__seq_remove_duplicates:NN #1#2
```

```
  {
    \seq_clear:N \l__seq_remove_seq
    \seq_map_inline:Nn #2
      {
        \seq_if_in:NnF \l__seq_remove_seq {##1}
          { \seq_put_right:Nn \l__seq_remove_seq {##1} }
      }
    #1 #2 \l__seq_remove_seq
  }
\cs_generate_variant:Nn \seq_remove_duplicates:N  { c }
\cs_generate_variant:Nn \seq_gremove_duplicates:N { c }
```

(*End of definition for* `\seq_remove_duplicates:N`*,* `\seq_gremove_duplicates:N`*, and* `\__seq_remove_duplicates:NN`*. These functions are documented on page 157.*)

`\seq_remove_all:Nn`
`\seq_remove_all:NV`
`\seq_remove_all:Ne`
`\seq_remove_all:Nx`
`\seq_remove_all:cn`
`\seq_remove_all:cV`
`\seq_remove_all:ce`
`\seq_remove_all:cx`
`\seq_gremove_all:Nn`
`\seq_gremove_all:NV`
`\seq_gremove_all:Ne`
`\seq_gremove_all:Nx`
`\seq_gremove_all:cn`
`\seq_gremove_all:cV`
`\seq_gremove_all:ce`
`\seq_gremove_all:Nx`
`\__seq_remove_all_aux:NNn`

The idea of the code here is to avoid a relatively expensive addition of items one at a time to an intermediate sequence. The approach taken is therefore similar to that in `\__seq_pop_right:NNN`, using a "flexible" `e`-type expansion to do most of the work. As `\tl_if_eq:nnT` is not expandable, a two-part strategy is needed. First, the `e`-type expansion uses `\str_if_eq:nnT` to find potential matches. If one is found, the expansion is halted and the necessary set up takes place to use the `\tl_if_eq:NNT` test. The `e`-type is started again, including all of the items copied already. This happens repeatedly until the entire sequence has been scanned. The code is set up to avoid needing an intermediate scratch list: the lead-off `e`-type expansion (`#1 #2 {#2}`) ensures that nothing is lost.

```
\cs_new_protected:Npn \seq_remove_all:Nn
  { \__seq_remove_all_aux:NNn \__kernel_tl_set:Nx }
\cs_new_protected:Npn \seq_gremove_all:Nn
  { \__seq_remove_all_aux:NNn \__kernel_tl_gset:Nx }
\cs_new_protected:Npn \__seq_remove_all_aux:NNn #1#2#3
  {
    \__seq_push_item_def:n
      {
        \str_if_eq:nnT {##1} {#3}
          {
            \if_false: { \fi: }
            \tl_set:Nn \l__seq_internal_b_tl {##1}
            #1 #2
              { \if_false: } \fi:
                \exp_not:o {#2}
                \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
                  { \use_none:nn }
          }
        \__seq_wrap_item:n {##1}
      }
    \tl_set:Nn \l__seq_internal_a_tl {#3}
    #1 #2 {#2}
    \__seq_pop_item_def:
  }
\cs_generate_variant:Nn \seq_remove_all:Nn  { NV , Ne , c , cV , ce }
\cs_generate_variant:Nn \seq_remove_all:Nn  { Nx , cx }
\cs_generate_variant:Nn \seq_gremove_all:Nn { NV , Ne , c , cV , ce }
\cs_generate_variant:Nn \seq_gremove_all:Nn { Nx , cx }
```

(*End of definition for* `\seq_remove_all:Nn`*,* `\seq_gremove_all:Nn`*, and* `\__seq_remove_all_aux:NNn`*. These functions are documented on page 157.*)

__seq_int_eval:w

Useful to more quickly go through items.

```
\cs_new_eq:NN \__seq_int_eval:w \tex_numexpr:D
```

(*End of definition for* `\__seq_int_eval:w`.)

\seq_set_item:Nnn
\seq_set_item:cnn
\seq_set_item:NnnTF
\seq_set_item:cnnTF
\seq_gset_item:Nnn
\seq_gset_item:cnn
\seq_gset_item:NnnTF
\seq_gset_item:cnnTF
__seq_set_item:NnnNN
__seq_set_item:nnNNNN
__seq_set_item_false:nnNNNN
__seq_set_item:nNnnNNNN
__seq_set_item:wn
__seq_set_item_end:w

The conditionals are distinguished from the `Nnn` versions by the last argument `\use_ii:nn` vs `\use_i:nn`.

```
\cs_new_protected:Npn \seq_set_item:Nnn #1#2#3
  { \__seq_set_item:NnnNN #1 {#2} {#3} \__kernel_tl_set:Nx \use_i:nn }
\cs_new_protected:Npn \seq_gset_item:Nnn #1#2#3
  { \__seq_set_item:NnnNN #1 {#2} {#3} \__kernel_tl_gset:Nx \use_i:nn }
\cs_generate_variant:Nn \seq_set_item:Nnn { c }
\cs_generate_variant:Nn \seq_gset_item:Nnn { c }
\prg_new_protected_conditional:Npnn \seq_set_item:Nnn #1#2#3 { TF , T , F }
  { \__seq_set_item:NnnNN #1 {#2} {#3} \__kernel_tl_set:Nx \use_ii:nn }
\prg_new_protected_conditional:Npnn \seq_gset_item:Nnn #1#2#3 { TF , T , F }
  { \__seq_set_item:NnnNN #1 {#2} {#3} \__kernel_tl_gset:Nx \use_ii:nn }
\prg_generate_conditional_variant:Nnn \seq_set_item:Nnn { c } { TF , T , F }
\prg_generate_conditional_variant:Nnn \seq_gset_item:Nnn { c } { TF , T , F }
```

Save the item to be stored and evaluate the position and the sequence length only once. Then depending on the sign of the position, check that it is not bigger than the length (in absolute value) nor zero.

```
\cs_new_protected:Npn \__seq_set_item:NnnNN #1#2#3
  {
    \tl_set:Nn \l__seq_internal_a_tl { \__seq_item:n {#3} }
    \exp_args:Nff \__seq_set_item:nnNNNN
      { \int_eval:n {#2} } { \seq_count:N #1 } #1 \use_none:nn
  }
\cs_new_protected:Npn \__seq_set_item:nnNNNN #1#2
  {
    \int_compare:nNnTF {#1} > 0
      { \int_compare:nNnF {#1} > {#2} { \__seq_set_item:nNnnNNNN { #1 - 1 } } }
      {
        \int_compare:nNnF {#1} < {-#2}
          {
            \int_compare:nNnF {#1} = 0
              { \__seq_set_item:nNnnNNNN { #2 + #1 } }
          }
      }
    \__seq_set_item_false:nnNNNN {#1} {#2}
  }
```

If the position is not ok, `\__seq_set_item_false:nnNNNN` calls an error or returns `false` (depending on the `\use_i:nn` vs `\use_ii:nn` argument mentioned above).

```
\cs_new_protected:Npn \__seq_set_item_false:nnNNNN #1#2#3#4#5#6
  {
    #6
      {
        \msg_error:nneee { seq } { item-too-large }
          { \token_to_str:N #3 } {#2} {#1}
      }
      { \prg_return_false: }
  }
```

If the position is ok, __seq_set_item:nNnnNNNN makes the assignment and returns true (in the case of conditionals). Here #1 is an integer expression (position minus one), it needs to be evaluated. The sequence #5 starts with \s__seq (even if empty), which stops the integer expression and is absorbed by it. The \if_meaning:w test is slightly faster than an integer test (but only works when testing against zero, hence the offset we chose in the position). When we are done skipping items, insert the saved item \l__-seq_internal_a_tl. For put functions the last argument of __seq_set_item_end:w is \use_none:nn and it absorbs the item #2 that we are removing: this is only useful for the pop functions.

```
\cs_new_protected:Npn \__seq_set_item:nNnnNNNN #1#2#3#4#5#6#7#8
  {
    #7 #5
      {
        \s__seq
        \exp_after:wN \__seq_set_item:wn
        \int_value:w \__seq_int_eval:w #1
        #5 \s__seq_stop #6
      }
    #8 { } { \prg_return_true: }
  }
\cs_new:Npn \__seq_set_item:wn #1 \__seq_item:n #2
  {
    \if_meaning:w 0 #1 \__seq_set_item_end:w \fi:
    \exp_not:n { \__seq_item:n {#2} }
    \exp_after:wN \__seq_set_item:wn
    \int_value:w \__seq_int_eval:w #1 - 1 \s__seq
  }
\cs_new:Npn \__seq_set_item_end:w #1 \exp_not:n #2 #3 \s__seq #4 \s__seq_stop #5
  {
    #1
    \exp_not:o \l__seq_internal_a_tl
    \exp_not:n {#4}
    #5 #2
  }
```

(*End of definition for* \seq_set_item:NnnTF *and others. These functions are documented on page 157.*)

\seq_reverse:N
\seq_reverse:c
\seq_greverse:N
\seq_greverse:c
__seq_reverse:NN
__seq_reverse_item:nwn

Previously, \seq_reverse:N was coded by collecting the items in reverse order after an \exp_stop_f: marker.

```
\cs_new_protected:Npn \seq_reverse:N #1
  {
    \cs_set_eq:NN \@@_item:n \@@_reverse_item:nw
    \tl_set:Nf #2 { #2 \exp_stop_f: }
  }
\cs_new:Npn \@@_reverse_item:nw #1 #2 \exp_stop_f:
  {
    #2 \exp_stop_f:
    \@@_item:n {#1}
  }
```

At first, this seems optimal, since we can forget about each item as soon as it is placed after \exp_stop_f:. Unfortunately, TeX's usual tail recursion does not take place in this case: since the following `\__seq_reverse_item:nw` only reads tokens until `\exp_-stop_f:`, and never reads the `\@@_item:n {#1}` left by the previous call, TeX cannot remove that previous call from the stack, and in particular must retain the various macro parameters in memory, until the end of the replacement text is reached. The stack is thus only flushed after all the `\__seq_reverse_item:nw` are expanded. Keeping track of the arguments of all those calls uses up a memory quadratic in the length of the sequence. TeX can then not cope with more than a few thousand items.

Instead, we collect the items in the argument of `\exp_not:n`. The previous calls are cleanly removed from the stack, and the memory consumption becomes linear.

```
\cs_new_protected:Npn \seq_reverse:N
  { \__seq_reverse:NN \__kernel_tl_set:Nx }
\cs_new_protected:Npn \seq_greverse:N
  { \__seq_reverse:NN \__kernel_tl_gset:Nx }
\cs_new_protected:Npn \__seq_reverse:NN #1 #2
  {
    \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
    \cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn
    #1 #2 { #2 \exp_not:n { } }
    \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
  }
\cs_new:Npn \__seq_reverse_item:nwn #1 #2 \exp_not:n #3
  {
    #2
    \exp_not:n { \__seq_item:n {#1} #3 }
  }
\cs_generate_variant:Nn \seq_reverse:N  { c }
\cs_generate_variant:Nn \seq_greverse:N { c }
```

(*End of definition for* `\seq_reverse:N` *and others. These functions are documented on page 157.*)

`\seq_sort:Nn`
`\seq_sort:cn`
`\seq_gsort:Nn`
`\seq_gsort:cn`

Implemented in l3sort.

(*End of definition for* `\seq_sort:Nn` *and* `\seq_gsort:Nn`*. These functions are documented on page 157.*)

58.4 Sequence conditionals

`\seq_if_empty_p:N`
`\seq_if_empty_p:c`
`\seq_if_empty:NTF`
`\seq_if_empty:cTF`

Similar to token lists, we compare with the empty sequence.

```
\prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \c_empty_seq
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_generate_conditional_variant:Nnn \seq_if_empty:N
  { c } { p , T , F , TF }
```

(*End of definition for* `\seq_if_empty:NTF`*. This function is documented on page 158.*)

`\seq_shuffle:N`
`\seq_shuffle:c`
`\seq_gshuffle:N`
`\seq_gshuffle:c`
`\__seq_shuffle:NN`
`\__seq_shuffle_item:n`
`\g__seq_internal_seq`

We apply the Fisher–Yates shuffle, storing items in `\toks` registers. We use the primitive `\tex_uniformdeviate:D` for speed reasons. Its non-uniformity is of order its argument divided by 2^{28}, not too bad for small lists. For sequences with more than 13 elements there are more possible permutations than possible seeds ($13! > 2^{28}$) so the question of uniformity is somewhat moot. The integer variables are declared in l3int: load-order issues.

```
\seq_new:N \g__seq_internal_seq
\cs_new_protected:Npn \seq_shuffle:N { \__seq_shuffle:NN \seq_set_eq:NN }
\cs_new_protected:Npn \seq_gshuffle:N { \__seq_shuffle:NN \seq_gset_eq:NN }
\cs_new_protected:Npn \__seq_shuffle:NN #1#2
  {
    \int_compare:nNnTF { \seq_count:N #2 } > \c_max_register_int
      {
        \msg_error:nne { seq } { shuffle-too-large }
          { \token_to_str:N #2 }
      }
      {
        \group_begin:
          \int_zero:N \l__seq_internal_a_int
          \__seq_push_item_def:
          \cs_gset_eq:NN \__seq_item:n \__seq_shuffle_item:n
          #2
          \__seq_pop_item_def:
          \seq_gclear:N \g__seq_internal_seq
          \int_step_inline:nn \l__seq_internal_a_int
            {
              \seq_gput_right:Ne \g__seq_internal_seq
                { \tex_the:D \tex_toks:D ##1 }
            }
        \group_end:
        #1 #2 \g__seq_internal_seq
        \seq_gclear:N \g__seq_internal_seq
    }
  }
\cs_new_protected:Npn \__seq_shuffle_item:n
  {
    \int_incr:N \l__seq_internal_a_int
    \int_set:Nn \l__seq_internal_b_int
      { 1 + \tex_uniformdeviate:D \l__seq_internal_a_int }
    \tex_toks:D \l__seq_internal_a_int
      = \tex_toks:D \l__seq_internal_b_int
    \tex_toks:D \l__seq_internal_b_int
  }
\cs_generate_variant:Nn \seq_shuffle:N { c }
\cs_generate_variant:Nn \seq_gshuffle:N { c }
```

(*End of definition for* `\seq_shuffle:N` *and others. These functions are documented on page* *158*.)

`\seq_if_in:NnTF`
`\seq_if_in:NVTF`
`\seq_if_in:NvTF`
`\seq_if_in:NeTF`
`\seq_if_in:NoTF`
`\seq_if_in:NxTF`
`\seq_if_in:cnTF`
`\seq_if_in:cVTF`
`\seq_if_in:cvTF`
`\seq_if_in:ceTF`
`\seq_if_in:coTF`
`\seq_if_in:cxTF`
`\__seq_if_in:`

The approach here is to define `\__seq_item:n` to compare its argument with the test sequence. If the two items are equal, the mapping is terminated and `\group_end: \prg_return_true:` is inserted after skipping over the rest of the recursion. On the other hand, if there is no match then the loop breaks, returning `\prg_return_false:`. Everything is inside a group so that `\__seq_item:n` is preserved in nested situations.

```
\prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
  { T , F , TF }
  {
    \group_begin:
```

```
      \tl_set:Nn \l__seq_internal_a_tl {#2}
      \cs_set_protected:Npn \__seq_item:n ##1
        {
          \tl_set:Nn \l__seq_internal_b_tl {##1}
          \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
            \exp_after:wN \__seq_if_in:
          \fi:
        }
      #1
    \group_end:
    \prg_return_false:
    \prg_break_point:
  }
\cs_new:Npn \__seq_if_in:
  { \prg_break:n { \group_end: \prg_return_true: } }
\prg_generate_conditional_variant:Nnn \seq_if_in:Nn
  { NV , Nv , Ne , No , Nx , c , cV , cv , ce , co , cx } { T , F , TF }
```

(*End of definition for* `\seq_if_in:NnTF` *and* `\__seq_if_in:`. *This function is documented on page 158.*)

58.5 Recovering data from sequences

`\__seq_pop:NNNN`
`\__seq_pop_TF:NNNN`

The two pop functions share their emptiness tests. We also use a common emptiness test for all branching `get` and `pop` functions.

```
\cs_new_protected:Npn \__seq_pop:NNNN #1#2#3#4
  {
    \if_meaning:w #3 \c_empty_seq
      \tl_set:Nn #4 { \q_no_value }
    \else:
      #1#2#3#4
    \fi:
  }
\cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
  {
    \if_meaning:w #3 \c_empty_seq
      % \tl_set:Nn #4 { \q_no_value }
      \prg_return_false:
    \else:
      #1#2#3#4
      \prg_return_true:
    \fi:
  }
```

(*End of definition for* `\__seq_pop:NNNN` *and* `\__seq_pop_TF:NNNN`.)

`\seq_get_left:NN`
`\seq_get_left:cN`
`\__seq_get_left:wnw`

Getting an item from the left of a sequence is pretty easy: just trim off the first item after `\__seq_item:n` at the start. We append a `\q_no_value` item to cover the case of an empty sequence

```
\cs_new_protected:Npn \seq_get_left:NN #1#2
  {
    \__kernel_tl_set:Nx #2
      {
        \exp_after:wN \__seq_get_left:wnw
```

```
      #1 \__seq_item:n { \q_no_value } \s__seq_stop
    }
  }
\cs_new:Npn \__seq_get_left:wnw #1 \__seq_item:n #2#3 \s__seq_stop
  { \exp_not:n {#2} }
\cs_generate_variant:Nn \seq_get_left:NN { c }
```

(*End of definition for* `\seq_get_left:NN` *and* `\__seq_get_left:wnw`*. This function is documented on page 154.*)

`\seq_pop_left:NN`
`\seq_pop_left:cN`
`\seq_gpop_left:NN`
`\seq_gpop_left:cN`
`\__seq_pop_left:NNN`
`\__seq_pop_left:wnwNNN`

The approach to popping an item is pretty similar to that to get an item, with the only difference being that the sequence itself has to be redefined. This makes it more sensible to use an auxiliary function for the local and global cases.

```
\cs_new_protected:Npn \seq_pop_left:NN
  { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_set:Nn }
\cs_new_protected:Npn \seq_gpop_left:NN
  { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_gset:Nn }
\cs_new_protected:Npn \__seq_pop_left:NNN #1#2#3
  { \exp_after:wN \__seq_pop_left:wnwNNN #2 \s__seq_stop #1#2#3 }
\cs_new_protected:Npn \__seq_pop_left:wnwNNN
    #1 \__seq_item:n #2#3 \s__seq_stop #4#5#6
  {
    #4 #5 { #1 #3 }
    \tl_set:Nn #6 {#2}
  }
\cs_generate_variant:Nn \seq_pop_left:NN  { c }
\cs_generate_variant:Nn \seq_gpop_left:NN { c }
```

(*End of definition for* `\seq_pop_left:NN` *and others. These functions are documented on page 155.*)

`\seq_get_right:NN`
`\seq_get_right:cN`
`\__seq_get_right_loop:nw`
`\__seq_get_right_end:NnN`

First remove `\s__seq` and prepend `\q_no_value`. The first argument of `\__seq_get_right_loop:nw` is the last item found, and the second argument is empty until the end of the loop, where it is code that applies `\exp_not:n` to the last item and ends the loop.

```
\cs_new_protected:Npn \seq_get_right:NN #1#2
  {
    \__kernel_tl_set:Nx #2
      {
        \exp_after:wN \use_i_ii:nnn
        \exp_after:wN \__seq_get_right_loop:nw
        \exp_after:wN \q_no_value
        #1
        \__seq_get_right_end:NnN \__seq_item:n
      }
  }
\cs_new:Npn \__seq_get_right_loop:nw #1#2 \__seq_item:n
  {
    #2 \use_none:n {#1}
    \__seq_get_right_loop:nw
  }
\cs_new:Npn \__seq_get_right_end:NnN #1#2#3 { \exp_not:n {#2} }
\cs_generate_variant:Nn \seq_get_right:NN { c }
```

(*End of definition for* `\seq_get_right:NN`*,* `\__seq_get_right_loop:nw`*, and* `\__seq_get_right_end:NnN`*. This function is documented on page 155.*)

\seq_pop_right:NN
\seq_pop_right:cN
\seq_gpop_right:NN
\seq_gpop_right:cN
__seq_pop_right:NNN
__seq_pop_right_loop:nn

The approach to popping from the right is a bit more involved, but does use some of the same ideas as getting from the right. What is needed is a "flexible length" way to set a token list variable. This is supplied by the `{ \if_false: } \fi:` ...`\if_false: { \fi: }` construct. Using an `e`-type expansion and a "non-expanding" definition for `\__seq_item:n`, the left-most $n-1$ entries in a sequence of n items are stored back in the sequence. That needs a loop of unknown length, hence using the strange `\if_false:` way of including braces. When the last item of the sequence is reached, the closing brace for the assignment is inserted, and `\tl_set:Nn #3` is inserted in front of the final entry. This therefore does the pop assignment. One more iteration is performed, with an empty argument and `\use_none:nn`, which finally stops the loop.

```
\cs_new_protected:Npn \seq_pop_right:NN
  { \__seq_pop:NNNN \__seq_pop_right:NNN \__kernel_tl_set:Nx }
\cs_new_protected:Npn \seq_gpop_right:NN
  { \__seq_pop:NNNN \__seq_pop_right:NNN \__kernel_tl_gset:Nx }
\cs_new_protected:Npn \__seq_pop_right:NNN #1#2#3
  {
    \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
    \cs_set_eq:NN \__seq_item:n \scan_stop:
    #1 #2
      { \if_false: } \fi: \s__seq
        \exp_after:wN \use_i:nnn
        \exp_after:wN \__seq_pop_right_loop:nn
        #2
        {
          \if_false: { \fi: }
          \__kernel_tl_set:Nx #3
        }
        { } \use_none:nn
    \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
  }
\cs_new:Npn \__seq_pop_right_loop:nn #1#2
  {
    #2 { \exp_not:n {#1} }
    \__seq_pop_right_loop:nn
  }
\cs_generate_variant:Nn \seq_pop_right:NN  { c }
\cs_generate_variant:Nn \seq_gpop_right:NN { c }
```

(*End of definition for* `\seq_pop_right:NN` *and others. These functions are documented on page* *155*.)

\seq_get_left:NN*TF*
\seq_get_left:cN*TF*
\seq_get_right:NN*TF*
\seq_get_right:cN*TF*

Getting from the left or right with a check on the results. The first argument to `\__seq_pop_TF:NNNN` is left unused.

```
\prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
  { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
\prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
  { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
\prg_generate_conditional_variant:Nnn \seq_get_left:NN
  { c } { T , F , TF }
\prg_generate_conditional_variant:Nnn \seq_get_right:NN
  { c } { T , F , TF }
```

(*End of definition for* `\seq_get_left:NNTF` *and* `\seq_get_right:NNTF`*. These functions are documented on page* *156*.)

\seq_pop_left:NNTF
\seq_pop_left:cNTF
\seq_gpop_left:NNTF
\seq_gpop_left:cNTF
\seq_pop_right:NNTF
\seq_pop_right:cNTF
\seq_gpop_right:NNTF
\seq_gpop_right:cNTF

More or less the same for popping.

```
\prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2
  { T , F , TF }
  { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_set:Nn #1 #2 }
\prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2
  { T , F , TF }
  { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_gset:Nn #1 #2 }
\prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2
  { T , F , TF }
  { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \__kernel_tl_set:Nx #1 #2 }
\prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2
  { T , F , TF }
  { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \__kernel_tl_gset:Nx #1 #2 }
\prg_generate_conditional_variant:Nnn \seq_pop_left:NN { c }
  { T , F , TF }
\prg_generate_conditional_variant:Nnn \seq_gpop_left:NN { c }
  { T , F , TF }
\prg_generate_conditional_variant:Nnn \seq_pop_right:NN { c }
  { T , F , TF }
\prg_generate_conditional_variant:Nnn \seq_gpop_right:NN { c }
  { T , F , TF }
```

(*End of definition for* `\seq_pop_left:NNTF` *and others. These functions are documented on page 156.*)

\seq_item:Nn
\seq_item:NV
\seq_item:Ne
\seq_item:cn
\seq_item:cV
\seq_item:ce
__seq_item:wNn
__seq_item:nN
__seq_item:nwn

The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then the argument delimited by `\__seq_item:n` is `\prg_break:` instead of being empty, terminating the loop and returning nothing at all.

```
\cs_new:Npn \seq_item:Nn #1
  { \exp_after:wN \__seq_item:wNn #1 \s__seq_stop #1 }
\cs_new:Npn \__seq_item:wNn \s__seq #1 \s__seq_stop #2#3
  {
    \exp_args:Nf \__seq_item:nwn
      { \exp_args:Nf \__seq_item:nN { \int_eval:n {#3} } #2 }
    #1
    \prg_break: \__seq_item:n { }
    \prg_break_point:
  }
\cs_new:Npn \__seq_item:nN #1#2
  {
    \int_compare:nNnTF {#1} < 0
      { \int_eval:n { \seq_count:N #2 + 1 + #1 } }
      {#1}
  }
\cs_new:Npn \__seq_item:nwn #1#2 \__seq_item:n #3
  {
    #2
    \int_compare:nNnTF {#1} = 1
      { \prg_break:n { \exp_not:n {#3} } }
      { \exp_args:Nf \__seq_item:nwn { \int_eval:n { #1 - 1 } } }
  }
\cs_generate_variant:Nn \seq_item:Nn { NV , Ne , c , cV , ce }
```

(*End of definition for* `\seq_item:Nn` *and others. This function is documented on page 155.*)

\seq_rand_item:N
\seq_rand_item:c

Importantly, \seq_item:Nn only evaluates its argument once.

```
\cs_new:Npn \seq_rand_item:N #1
  {
    \seq_if_empty:NF #1
      { \seq_item:Nn #1 { \int_rand:nn { 1 } { \seq_count:N #1 } } }
  }
\cs_generate_variant:Nn \seq_rand_item:N { c }
```

(*End of definition for* \seq_rand_item:N*. This function is documented on page 155.*)

58.6 Mapping over sequences

\seq_map_break:
\seq_map_break:n

To break a function, the special token \prg_break_point:Nn is used to find the end of the code. Any ending code is then inserted before the return value of \seq_map_break:n is inserted.

```
\cs_new:Npn \seq_map_break:
  { \prg_map_break:Nn \seq_map_break: { } }
\cs_new:Npn \seq_map_break:n
  { \prg_map_break:Nn \seq_map_break: }
```

(*End of definition for* \seq_map_break: *and* \seq_map_break:n*. These functions are documented on page 160.*)

\seq_map_function:NN
\seq_map_function:cN
__seq_map_function:Nw

The idea here is to apply the code of #2 to each item in the sequence without altering the definition of __seq_item:n. The even-numbered arguments of __seq_map_function:Nw delimited by __seq_item:n are almost always empty, except at the end of the loop where it is \prg_break:. This allows to break the loop without needing to do a (relatively-expensive) quark test.

```
\cs_new:Npn \seq_map_function:NN #1#2
  {
    \exp_after:wN \use_i_ii:nnn
    \exp_after:wN \__seq_map_function:Nw
    \exp_after:wN #2
    #1
    \prg_break:
    \__seq_item:n { } \__seq_item:n { } \__seq_item:n { } \__seq_item:n { }
    \prg_break_point:
    \prg_break_point:Nn \seq_map_break: { }
  }
\cs_new:Npn \__seq_map_function:Nw #1
    #2 \__seq_item:n #3
    #4 \__seq_item:n #5
    #6 \__seq_item:n #7
    #8 \__seq_item:n #9
  {
    #2 #1 {#3}
    #4 #1 {#5}
    #6 #1 {#7}
    #8 #1 {#9}
    \__seq_map_function:Nw #1
  }
\cs_generate_variant:Nn \seq_map_function:NN { c }
```

(*End of definition for* `\seq_map_function:NN` *and* `\__seq_map_function:Nw`. *This function is documented on page 158.*)

`\__seq_push_item_def:n`
`\__seq_push_item_def:e`
`\__seq_push_item_def:`
`\__seq_pop_item_def:`

The definition of `\__seq_item:n` needs to be saved and restored at various points within the mapping and manipulation code. That is handled here: as always, this approach uses global assignments.

```
\cs_new_protected:Npn \__seq_push_item_def:n
  {
    \__seq_push_item_def:
    \cs_gset:Npn \__seq_item:n ##1
  }
\cs_new_protected:Npn \__seq_push_item_def:e
  {
    \__seq_push_item_def:
    \cs_gset:Npe \__seq_item:n ##1
  }
\cs_new_protected:Npn \__seq_push_item_def:
  {
    \int_gincr:N \g__kernel_prg_map_int
    \cs_gset_eq:cN { __seq_map_ \int_use:N \g__kernel_prg_map_int :w }
      \__seq_item:n
  }
\cs_new_protected:Npn \__seq_pop_item_def:
  {
    \cs_gset_eq:Nc \__seq_item:n
      { __seq_map_ \int_use:N \g__kernel_prg_map_int :w }
    \int_gdecr:N \g__kernel_prg_map_int
  }
```

(*End of definition for* `\__seq_push_item_def:n`, `\__seq_push_item_def:`, *and* `\__seq_pop_item_def:`.)

`\seq_map_inline:Nn`
`\seq_map_inline:cn`

The idea here is that `\__seq_item:n` is already "applied" to each item in a sequence, and so an in-line mapping is just a case of redefining `\__seq_item:n`.

```
\cs_new_protected:Npn \seq_map_inline:Nn #1#2
  {
    \__seq_push_item_def:n {#2}
    #1
    \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
  }
\cs_generate_variant:Nn \seq_map_inline:Nn { c }
```

(*End of definition for* `\seq_map_inline:Nn`. *This function is documented on page 158.*)

`\seq_map_tokens:Nn`
`\seq_map_tokens:cn`
`\__seq_map_tokens:nw`

This is based on the function mapping but using the same tricks as described for `\prop_map_tokens:Nn`. The idea is to remove the leading `\s__seq` and apply the tokens such that they are safe with the break points, hence the `\use:n`.

```
\cs_new:Npn \seq_map_tokens:Nn #1#2
  {
    \exp_last_unbraced:Nno
      \use_i:nn { \__seq_map_tokens:nw {#2} } #1
    \prg_break:
    \__seq_item:n { } \__seq_item:n { } \__seq_item:n { } \__seq_item:n { }
    \prg_break_point:
    \prg_break_point:Nn \seq_map_break: { }
```

```
  }
\cs_generate_variant:Nn \seq_map_tokens:Nn { c }
\cs_new:Npn \__seq_map_tokens:nw #1
    #2 \__seq_item:n #3
    #4 \__seq_item:n #5
    #6 \__seq_item:n #7
    #8 \__seq_item:n #9
  {
    #2 \use:n {#1} {#3}
    #4 \use:n {#1} {#5}
    #6 \use:n {#1} {#7}
    #8 \use:n {#1} {#9}
    \__seq_map_tokens:nw {#1}
  }
```

(*End of definition for* `\seq_map_tokens:Nn` *and* `\__seq_map_tokens:nw`. *This function is documented on page* *159*.)

`\seq_map_variable:NNn`
`\seq_map_variable:Ncn`
`\seq_map_variable:cNn`
`\seq_map_variable:ccn`

This is just a specialised version of the in-line mapping function, using an `e`-type expansion for the code set up so that the number of `#` tokens required is as expected.

```
\cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
  {
    \__seq_push_item_def:e
      {
        \tl_set:Nn \exp_not:N #2 {##1}
        \exp_not:n {#3}
      }
    #1
    \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
  }
\cs_generate_variant:Nn \seq_map_variable:NNn {     Nc }
\cs_generate_variant:Nn \seq_map_variable:NNn { c , cc }
```

(*End of definition for* `\seq_map_variable:NNn`. *This function is documented on page* *159*.)

`\seq_map_indexed_function:NN`
`\seq_map_indexed_inline:Nn`
`\__seq_map_indexed:nNN`
`\__seq_map_indexed:Nw`

Similar to `\seq_map_function:NN` but we keep track of the item index as a `;`-delimited argument of `\__seq_map_indexed:Nw`.

```
\cs_new:Npn \seq_map_indexed_function:NN #1#2
  {
    \__seq_map_indexed:NN #1#2
    \prg_break_point:Nn \seq_map_break: { }
  }
\cs_new_protected:Npn \seq_map_indexed_inline:Nn #1#2
  {
    \int_gincr:N \g__kernel_prg_map_int
    \cs_gset_protected:cpn
      { __seq_map_ \int_use:N \g__kernel_prg_map_int :w } ##1##2 {#2}
    \exp_args:NNc \__seq_map_indexed:NN #1
      { __seq_map_ \int_use:N \g__kernel_prg_map_int :w }
    \prg_break_point:Nn \seq_map_break:
      { \int_gdecr:N \g__kernel_prg_map_int }
  }
\cs_new:Npn \__seq_map_indexed:NN #1#2
  {
```

```
    \exp_after:wN \__seq_map_indexed:Nw
    \exp_after:wN #2
    \int_value:w 1
    \exp_after:wN \use_i:nn
    \exp_after:wN ;
    #1
    \prg_break: \__seq_item:n { } \prg_break_point:
  }
\cs_new:Npn \__seq_map_indexed:Nw #1#2 ; #3 \__seq_item:n #4
  {
    #3
    #1 {#2} {#4}
    \exp_after:wN \__seq_map_indexed:Nw
    \exp_after:wN #1
    \int_value:w \int_eval:w 1 + #2 ;
  }
```

(*End of definition for* `\seq_map_indexed_function:NN` *and others. These functions are documented on page* *159*.)

\seq_map_pairwise_function:NNN
\seq_map_pairwise_function:NcN
\seq_map_pairwise_function:cNN
\seq_map_pairwise_function:ccN
__seq_map_pairwise_function:wNN
__seq_map_pairwise_function:wNw
__seq_map_pairwise_function:Nnnwnn

The idea is to first expand both sequences, adding the usual `{ ? \prg_break: } { }` to the end of each one. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first tokens of `#2` and `#5`, which for items in both sequences are `\s__seq \__seq_item:n`. The function to be mapped are then be applied to the two entries. When the code hits the end of one of the sequences, the break material stops the entire loop and tidy up. This avoids needing to find the count of the two sequences, or worrying about which is longer.

```
\cs_new:Npn \seq_map_pairwise_function:NNN #1#2#3
  { \exp_after:wN \__seq_map_pairwise_function:wNN #2 \s__seq_stop #1 #3 }
\cs_new:Npn \__seq_map_pairwise_function:wNN \s__seq #1 \s__seq_stop #2#3
  {
    \exp_after:wN \__seq_map_pairwise_function:wNw #2 \s__seq_stop #3
      #1 { ? \prg_break: } { }
    \prg_break_point:
    \prg_break_point:Nn \seq_map_break: { }
  }
\cs_new:Npn \__seq_map_pairwise_function:wNw \s__seq #1 \s__seq_stop #2
  {
    \__seq_map_pairwise_function:Nnnwnn #2
      #1 { ? \prg_break: } { }
    \s__seq_stop
  }
\cs_new:Npn \__seq_map_pairwise_function:Nnnwnn #1#2#3#4 \s__seq_stop #5#6
  {
    \use_none:n #2
    \use_none:n #5
    #1 {#3} {#6}
    \__seq_map_pairwise_function:Nnnwnn #1 #4 \s__seq_stop
  }
\cs_generate_variant:Nn \seq_map_pairwise_function:NNN { Nc , c , cc }
```

(*End of definition for* `\seq_map_pairwise_function:NNN` *and others. This function is documented on page* *159*.)

\seq_set_map_e:NNn
\seq_gset_map_e:NNn
__seq_set_map_e:NNNn

Very similar to \seq_set_filter:NNn. We could actually merge the two within a single function, but it would have weird semantics.

```
\cs_new_protected:Npn \seq_set_map_e:NNn
  { \__seq_set_map_e:NNNn \__kernel_tl_set:Nx }
\cs_new_protected:Npn \seq_gset_map_e:NNn
  { \__seq_set_map_e:NNNn \__kernel_tl_gset:Nx }
\cs_new_protected:Npn \__seq_set_map_e:NNNn #1#2#3#4
  {
    \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
    #1 #2 { #3 }
    \__seq_pop_item_def:
  }
```

(*End of definition for* \seq_set_map_e:NNn *,* \seq_gset_map_e:NNn *, and* __seq_set_map_e:NNNn*. These functions are documented on page 161.*)

\seq_set_map:NNn
\seq_gset_map:NNn
__seq_set_map:NNNn

Similar to \seq_set_map_e:NNn, but prevents expansion of the <inline function>.

```
\cs_new_protected:Npn \seq_set_map:NNn
  { \__seq_set_map:NNNn \__kernel_tl_set:Nx }
\cs_new_protected:Npn \seq_gset_map:NNn
  { \__seq_set_map:NNNn \__kernel_tl_gset:Nx }
\cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
  {
    \__seq_push_item_def:n { \exp_not:n { \__seq_item:n {#4} } }
    #1 #2 { #3 }
    \__seq_pop_item_def:
  }
```

(*End of definition for* \seq_set_map:NNn *,* \seq_gset_map:NNn *, and* __seq_set_map:NNNn*. These functions are documented on page 160.*)

\seq_count:N
\seq_count:c
__seq_count:w
__seq_count_end:w

Since counting the items in a sequence is quite common, we optimize it by grabbing 8 items at a time and correspondingly adding 8 to an integer expression. At the end of the loop, #9 is __seq_count_end:w instead of being empty. It removes 8+ and instead places the number of __seq_item:n that __seq_count:w grabbed before reaching the end of the sequence.

```
\cs_new:Npn \seq_count:N #1
  {
    \int_eval:n
      {
        \exp_after:wN \use_i:nn
        \exp_after:wN \__seq_count:w
        #1
        \__seq_count_end:w \__seq_item:n 7
        \__seq_count_end:w \__seq_item:n 6
        \__seq_count_end:w \__seq_item:n 5
        \__seq_count_end:w \__seq_item:n 4
        \__seq_count_end:w \__seq_item:n 3
        \__seq_count_end:w \__seq_item:n 2
        \__seq_count_end:w \__seq_item:n 1
        \__seq_count_end:w \__seq_item:n 0
        \prg_break_point:
      }
  }
```

```
\cs_new:Npn \__seq_count:w
    #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4 \__seq_item:n
    #5 \__seq_item:n #6 \__seq_item:n #7 \__seq_item:n #8 #9 \__seq_item:n
  { #9 8 + \__seq_count:w }
\cs_new:Npn \__seq_count_end:w 8 + \__seq_count:w #1#2 \prg_break_point: {#1}
\cs_generate_variant:Nn \seq_count:N { c }
```

(*End of definition for* `\seq_count:N`*,* `\__seq_count:w`*, and* `\__seq_count_end:w`*. This function is documented on page* 161*.*)

58.7 Using sequences

`\seq_use:Nnnn`
`\seq_use:cnnn`
`\__seq_use:NNnNnn`
`\__seq_use_setup:w`
`\__seq_use:nwwwwnwn`
`\__seq_use:nwwn`
`\seq_use:Nn`
`\seq_use:cn`

See `\clist_use:Nnnn` for a general explanation. The main difference is that we use `\__seq_item:n` as a delimiter rather than commas. We also need to add `\__seq_item:n` at various places, and `\s__seq`.

```
\cs_new:Npn \seq_use:Nnnn #1#2#3#4
  {
    \seq_if_exist:NTF #1
      {
        \int_case:nnF { \seq_count:N #1 }
          {
            { 0 } { }
            { 1 } { \exp_after:wN \__seq_use:NNnNnn #1 ? { } { } }
            { 2 } { \exp_after:wN \__seq_use:NNnNnn #1 {#2} }
          }
          {
            \exp_after:wN \__seq_use_setup:w #1 \__seq_item:n
            \s__seq_mark { \__seq_use:nwwwwnwn {#3} }
            \s__seq_mark { \__seq_use:nwwn {#4} }
            \s__seq_stop { }
          }
      }
      {
        \msg_expandable_error:nnn
          { kernel } { bad-variable } {#1}
      }
  }
\cs_generate_variant:Nn \seq_use:Nnnn { c }
\cs_new:Npn \__seq_use:NNnNnn #1#2#3#4#5#6 { \exp_not:n { #3 #6 #5 } }
\cs_new:Npn \__seq_use_setup:w \s__seq { \__seq_use:nwwwwnwn { } }
\cs_new:Npn \__seq_use:nwwwwnwn
    #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4#5
    \s__seq_mark #6#7 \s__seq_stop #8
  {
    #6 \__seq_item:n {#3} \__seq_item:n {#4} #5
    \s__seq_mark {#6} #7 \s__seq_stop { #8 #1 #2 }
  }
\cs_new:Npn \__seq_use:nwwn #1 \__seq_item:n #2 #3 \s__seq_stop #4
  { \exp_not:n { #4 #1 #2 } }
\cs_new:Npn \seq_use:Nn #1#2
  { \seq_use:Nnnn #1 {#2} {#2} {#2} }
\cs_generate_variant:Nn \seq_use:Nn { c }
```

(*End of definition for* `\seq_use:Nnnn` *and others. These functions are documented on page* 161*.*)

58.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

\seq_push:Nn
\seq_push:NV
\seq_push:Nv
\seq_push:Ne
\seq_push:No
\seq_push:Nx
\seq_push:cn
\seq_push:cV
\seq_push:cv
\seq_push:ce
\seq_push:co
\seq_push:cx
\seq_gpush:Nn
\seq_gpush:NV
\seq_gpush:Nv
\seq_gpush:Ne
\seq_gpush:No
\seq_gpush:Nx
\seq_gpush:cn
\seq_gpush:cV
\seq_gpush:cv
\seq_gpush:ce
\seq_gpush:co
\seq_gpush:cx

Pushing to a sequence is the same as adding on the left.

```
\cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
\cs_generate_variant:Nn \seq_push:Nn { NV , Nv , Ne , c , cV , cv , ce }
\cs_generate_variant:Nn \seq_push:Nn { No , Nx , co , cx }
\cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
\cs_generate_variant:Nn \seq_gpush:Nn { NV , Nv , Ne , c , cV , cv , ce }
\cs_generate_variant:Nn \seq_gpush:Nn { No , Nx , co , cx }
```

(*End of definition for* \seq_push:Nn *and* \seq_gpush:Nn*. These functions are documented on page* 163*.*)

\seq_get:NN
\seq_get:cN
\seq_pop:NN
\seq_pop:cN
\seq_gpop:NN
\seq_gpop:cN

In most cases, getting items from the stack does not need to specify that this is from the left. So alias are provided.

```
\cs_new_eq:NN \seq_get:NN \seq_get_left:NN
\cs_new_eq:NN \seq_get:cN \seq_get_left:cN
\cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
\cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
\cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
\cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN
```

(*End of definition for* \seq_get:NN *,* \seq_pop:NN *, and* \seq_gpop:NN*. These functions are documented on page* 162*.*)

\seq_get:NNTF
\seq_get:cNTF
\seq_pop:NNTF
\seq_pop:cNTF
\seq_gpop:NNTF
\seq_gpop:cNTF

More copies.

```
\prg_new_eq_conditional:NNn \seq_get:NN  \seq_get_left:NN  { T , F , TF }
\prg_new_eq_conditional:NNn \seq_get:cN  \seq_get_left:cN  { T , F , TF }
\prg_new_eq_conditional:NNn \seq_pop:NN  \seq_pop_left:NN  { T , F , TF }
\prg_new_eq_conditional:NNn \seq_pop:cN  \seq_pop_left:cN  { T , F , TF }
\prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
\prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }
```

(*End of definition for* \seq_get:NNTF *,* \seq_pop:NNTF *, and* \seq_gpop:NNTF*. These functions are documented on page* 162*.*)

58.9 Viewing sequences

\seq_show:N
\seq_show:c
\seq_log:N
\seq_log:c
__seq_show:NN
__seq_show_validate:nn

Apply the general __kernel_chk_tl_type:NnnT.

```
\cs_new_protected:Npn \seq_show:N { \__seq_show:NN \msg_show:nneeee }
\cs_generate_variant:Nn \seq_show:N { c }
\cs_new_protected:Npn \seq_log:N { \__seq_show:NN \msg_log:nneeee }
\cs_generate_variant:Nn \seq_log:N { c }
\cs_new_protected:Npn \__seq_show:NN #1#2
  {
    \__kernel_chk_tl_type:NnnT #2 { seq }
      {
        \s__seq
        \exp_after:wN \use_i:nn \exp_after:wN \__seq_show_validate:nn #2
        \q_recursion_tail \q_recursion_tail \q_recursion_stop
      }
      {
        #1 { seq } { show }
```

```
            { \token_to_str:N #2 }
            { \seq_map_function:NN #2 \msg_show_item:n }
            { } { }
        }
    }
\cs_new:Npn \__seq_show_validate:nn #1#2
  {
    \quark_if_recursion_tail_stop:n {#2}
    \__seq_wrap_item:n {#2}
    \__seq_show_validate:nn
  }
```

(*End of definition for* `\seq_show:N` *and others. These functions are documented on page* 165.)

58.10 Scratch sequences

`\l_tmpa_seq`
`\l_tmpb_seq`
`\g_tmpa_seq`
`\g_tmpb_seq`

Temporary comma list variables.

```
\seq_new:N \l_tmpa_seq
\seq_new:N \l_tmpb_seq
\seq_new:N \g_tmpa_seq
\seq_new:N \g_tmpb_seq
```

(*End of definition for* `\l_tmpa_seq` *and others. These variables are documented on page* 165.)

```
⟨/package⟩
```

Chapter 59

l3int implementation

```
⟨*package⟩
⟨@@=int⟩
```

The following test files are used for this code: m3int001,m3int002,m3int03.

\c_max_register_int Done in l3basics.

(*End of definition for* \c_max_register_int*. This variable is documented on page 177.*)

__int_to_roman:w
\if_int_compare:w

Done in l3basics.

(*End of definition for* __int_to_roman:w *and* \if_int_compare:w*. This function is documented on page 178.*)

\or: Done in l3basics.

(*End of definition for* \or:*. This function is documented on page 178.*)

\int_value:w
__int_eval:w
__int_eval_end:
\if_int_odd:w
\if_case:w

Here are the remaining primitives for number comparisons and expressions.

```
\cs_new_eq:NN \int_value:w      \tex_number:D
\cs_new_eq:NN \__int_eval:w      \tex_numexpr:D
\cs_new_eq:NN \__int_eval_end:   \tex_relax:D
\cs_new_eq:NN \if_int_odd:w     \tex_ifodd:D
\cs_new_eq:NN \if_case:w        \tex_ifcase:D
```

(*End of definition for* \int_value:w *and others. These functions are documented on page 178.*)

\s__int_mark
\s__int_stop

Scan marks used throughout the module.

```
\scan_new:N \s__int_mark
\scan_new:N \s__int_stop
```

(*End of definition for* \s__int_mark *and* \s__int_stop*.*)

__int_use_none_delimit_by_s_stop:w Function to gobble until a scan mark.

```
\cs_new:Npn \__int_use_none_delimit_by_s_stop:w #1 \s__int_stop { }
```

(*End of definition for* __int_use_none_delimit_by_s_stop:w*.*)

\q__int_recursion_tail
\q__int_recursion_stop

Quarks for recursion.

```
\quark_new:N \q__int_recursion_tail
\quark_new:N \q__int_recursion_stop
```

(*End of definition for* `\q__int_recursion_tail` *and* `\q__int_recursion_stop`.)

`\__int_if_recursion_tail_stop_do:Nn`
`\__int_if_recursion_tail_stop:N`

Functions to query quarks.

```
\__kernel_quark_new_test:N \__int_if_recursion_tail_stop_do:Nn
\__kernel_quark_new_test:N \__int_if_recursion_tail_stop:N
```

(*End of definition for* `\__int_if_recursion_tail_stop_do:Nn` *and* `\__int_if_recursion_tail_stop:N`.)

59.1 Integer expressions

`\int_eval:n`
`\int_eval:w`

Wrapper for `\__int_eval:w`: can be used in an integer expression or directly in the input stream. It is very slightly faster to use `\the` rather than `\number` to turn the expression to a number. When debugging, we introduce parentheses to catch early termination (see l3debug).

```
\cs_new:Npn \int_eval:n #1
  { \tex_the:D \__int_eval:w #1 \__int_eval_end: }
\cs_new:Npn \int_eval:w { \tex_the:D \__int_eval:w }
```

(*End of definition for* `\int_eval:n` *and* `\int_eval:w`. *These functions are documented on page 167.*)

`\int_sign:n`
`\__int_sign:Nw`

See `\int_abs:n`. Evaluate the expression once (and when debugging is enabled, check that the expression is well-formed), then test the first character to determine the sign. This is wrapped in `\int_value:w` … `\exp_stop_f:` to ensure a fixed number of expansions and to avoid dealing with closing the conditionals.

```
\cs_new:Npn \int_sign:n #1
  {
    \int_value:w \exp_after:wN \__int_sign:Nw
      \int_value:w \__int_eval:w #1 \__int_eval_end: ;
    \exp_stop_f:
  }
\cs_new:Npn \__int_sign:Nw #1#2 ;
  {
    \if_meaning:w 0 #1
      0
    \else:
      \if_meaning:w - #1 - \fi: 1
    \fi:
  }
```

(*End of definition for* `\int_sign:n` *and* `\__int_sign:Nw`. *This function is documented on page 167.*)

`\int_abs:n`
`\__int_abs:N`
`\int_max:nn`
`\int_min:nn`
`\__int_maxmin:wwN`

Functions for min, max, and absolute value with only one evaluation. The absolute value is obtained by removing a leading sign if any. All three functions expand in two steps.

```
\cs_new:Npn \int_abs:n #1
  {
    \int_value:w \exp_after:wN \__int_abs:N
      \int_value:w \__int_eval:w #1 \__int_eval_end:
    \exp_stop_f:
  }
\cs_new:Npn \__int_abs:N #1
  { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
\cs_new:Npn \int_max:nn #1#2
  {
```

```
    \int_value:w \exp_after:wN \__int_maxmin:wwN
      \int_value:w \__int_eval:w #1 \exp_after:wN ;
      \int_value:w \__int_eval:w #2 ;
      >
    \exp_stop_f:
  }
\cs_new:Npn \int_min:nn #1#2
  {
    \int_value:w \exp_after:wN \__int_maxmin:wwN
      \int_value:w \__int_eval:w #1 \exp_after:wN ;
      \int_value:w \__int_eval:w #2 ;
      <
    \exp_stop_f:
  }
\cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
  {
    \if_int_compare:w #1 #3 #2 ~
      #1
    \else:
      #2
    \fi:
  }
```

(*End of definition for* `\int_abs:n` *and others. These functions are documented on page 167.*)

`\int_div_truncate:nn`
`\int_div_round:nn`
`\int_mod:nn`
`\__int_div_truncate:NwNw`
`\__int_mod:ww`

As `\__int_eval:w` rounds the result of a division we also provide a version that truncates the result. We use an auxiliary to make sure numerator and denominator are only evaluated once: this comes in handy when those are more expressions are expensive to evaluate (*e.g.*, `\tl_count:n`). If the numerator `#1#2` is 0, then we divide 0 by the denominator (this ensures that 0/0 is correctly reported as an error). Otherwise, shift the numerator `#1#2` towards 0 by $(|\texttt{#3#4}|-1)/2$, which we round away from zero. It turns out that this quantity exactly compensates the difference between ε-TeX's rounding and the truncating behaviour that we want. The details are thanks to Heiko Oberdiek: getting things right in all cases is not so easy.

```
\cs_new:Npn \int_div_truncate:nn #1#2
  {
    \int_value:w \__int_eval:w
      \exp_after:wN \__int_div_truncate:NwNw
      \int_value:w \__int_eval:w #1 \exp_after:wN ;
      \int_value:w \__int_eval:w #2 ;
    \__int_eval_end:
  }
\cs_new:Npn \__int_div_truncate:NwNw #1#2; #3#4;
  {
    \if_meaning:w 0 #1
      0
    \else:
      (
        #1#2
        \if_meaning:w - #1 + \else: - \fi:
        ( \if_meaning:w - #3 - \fi: #3#4 - 1 ) / 2
      )
    \fi:
    / #3#4
```

```
  }
```

For the sake of completeness:

```
\cs_new:Npn \int_div_round:nn #1#2
  { \int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }
```

Finally there's the modulus operation.

```
\cs_new:Npn \int_mod:nn #1#2
  {
    \int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
      \int_value:w \__int_eval:w #1 \exp_after:wN ;
      \int_value:w \__int_eval:w #2 ;
    \__int_eval_end:
  }
\cs_new:Npn \__int_mod:ww #1; #2;
  { #1 - ( \__int_div_truncate:NwNw #1 ; #2 ; ) * #2 }
```

(*End of definition for* `\int_div_truncate:nn` *and others. These functions are documented on page 167.*)

`\__kernel_int_add:nnn`

Equivalent to `\int_eval:n {#1+#2+#3}` except that overflow only occurs if the final result overflows $[-2^{31}+1, 2^{31}-1]$. The idea is to choose the order in which the three numbers are added together. If `#1` and `#2` have opposite signs (one is in $[-2^{31}+1, -1]$ and the other in $[0, 2^{31}-1]$) then `#1+#2` cannot overflow so we compute the result as `#1+#2+#3`. If they have the same sign, then either `#3` has the same sign and the order does not matter, or `#3` has the opposite sign and any order in which `#3` is not last will work. We use `#1+#3+#2`.

```
\cs_new:Npn \__kernel_int_add:nnn #1#2#3
  {
    \int_value:w \__int_eval:w #1
      \if_int_compare:w #2 < \c_zero_int \exp_after:wN \reverse_if:N \fi:
      \if_int_compare:w #1 < \c_zero_int + #2 + #3 \else: + #3 + #2 \fi:
    \__int_eval_end:
  }
```

(*End of definition for* `\__kernel_int_add:nnn`.)

59.2 Creating and initialising integers

`\int_new:N`
`\int_new:c`

Two ways to do this: one for the format and one for the LaTeX 2_ε package. In plain TeX, `\newcount` (and other allocators) are `\outer`: to allow the code here to work in "generic" mode this is therefore accessed by name. (The same applies to `\newbox`, `\newdimen` and so on.)

```
\cs_new_protected:Npn \int_new:N #1
  {
    \__kernel_chk_if_free_cs:N #1
    \cs:w newcount \cs_end: #1
  }
\cs_generate_variant:Nn \int_new:N { c }
```

(*End of definition for* `\int_new:N`. *This function is documented on page 168.*)

\int_const:Nn
\int_const:cn
__int_const:nN
__int_const:eN
__int_constdef:Nw
\c__int_max_constdef_int

As stated, most constants can be defined as `\chardef` or `\mathchardef` but that's engine dependent. As a result, there is some set up code to determine what can be done. No full engine testing just yet so everything is a little awkward. We cannot use `\int_gset:Nn` because (when `check-declarations` is enabled) this runs some checks that constants would fail.

```
\cs_new_protected:Npn \int_const:Nn #1#2
  { \__int_const:eN { \int_eval:n {#2} } #1 }
\cs_generate_variant:Nn \int_const:Nn { c }
\cs_new_protected:Npn \__int_const:nN #1#2
  {
    \int_compare:nNnTF {#1} < \c_zero_int
      {
        \int_new:N #2
        \tex_global:D
      }
      {
        \int_compare:nNnTF {#1} > \c__int_max_constdef_int
          {
            \int_new:N #2
            \tex_global:D
          }
          {
            \__kernel_chk_if_free_cs:N #2
            \tex_global:D \__int_constdef:Nw
          }
      }
    #2 = \__int_eval:w #1 \__int_eval_end:
  }
\cs_generate_variant:Nn \__int_const:nN { e }
\if_int_odd:w 0
  \cs_if_exist:NT \tex_luatexversion:D { 1 }
  \cs_if_exist:NT \tex_omathchardef:D  { 1 }
  \cs_if_exist:NT \tex_XeTeXversion:D  { 1 } ~
    \cs_if_exist:NTF \tex_omathchardef:D
      { \cs_new_eq:NN \__int_constdef:Nw \tex_omathchardef:D }
      { \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D }
    \tex_global:D \__int_constdef:Nw \c__int_max_constdef_int 1114111 ~
\else:
  \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
  \tex_global:D \__int_constdef:Nw \c__int_max_constdef_int 32767 ~
\fi:
```

(*End of definition for* `\int_const:Nn` *and others. This function is documented on page 168.*)

\int_zero:N
\int_zero:c
\int_gzero:N
\int_gzero:c

Functions that reset an ⟨*integer*⟩ register to zero.

```
\cs_new_protected:Npn \int_zero:N  #1 { #1 = \c_zero_int }
\cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero_int }
\cs_generate_variant:Nn \int_zero:N  { c }
\cs_generate_variant:Nn \int_gzero:N { c }
```

(*End of definition for* `\int_zero:N` *and* `\int_gzero:N`. *These functions are documented on page 168.*)

\int_zero_new:N
\int_zero_new:c
\int_gzero_new:N
\int_gzero_new:c

Create a register if needed, otherwise clear it.

```
\cs_new_protected:Npn \int_zero_new:N  #1
```

```
  { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }
\cs_new_protected:Npn \int_gzero_new:N #1
  { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }
\cs_generate_variant:Nn \int_zero_new:N  { c }
\cs_generate_variant:Nn \int_gzero_new:N { c }
```

(*End of definition for* `\int_zero_new:N` *and* `\int_gzero_new:N`. *These functions are documented on page 168.*)

`\int_set_eq:NN` `\int_set_eq:cN` `\int_set_eq:Nc` `\int_set_eq:cc` `\int_gset_eq:NN` `\int_gset_eq:cN` `\int_gset_eq:Nc` `\int_gset_eq:cc`

Setting equal means using one integer inside the set function of another. Check that assigned integer is local/global. No need to check that the other one is defined as TeX does it for us.

```
\cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }
\cs_generate_variant:Nn \int_set_eq:NN { c , Nc , cc }
\cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\cs_generate_variant:Nn \int_gset_eq:NN { c , Nc , cc }
```

(*End of definition for* `\int_set_eq:NN` *and* `\int_gset_eq:NN`. *These functions are documented on page 168.*)

`\int_if_exist_p:N` `\int_if_exist_p:c` `\int_if_exist:NTF` `\int_if_exist:cTF`

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \int_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \int_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End of definition for* `\int_if_exist:NTF`. *This function is documented on page 168.*)

59.3 Setting and incrementing integers

`\int_add:Nn` `\int_add:cn` `\int_gadd:Nn` `\int_gadd:cn` `\int_sub:Nn` `\int_sub:cn` `\int_gsub:Nn` `\int_gsub:cn`

Adding and subtracting to and from a counter. Including here the optional `by` would slow down these operations by a few percent.

```
\cs_new_protected:Npn \int_add:Nn #1#2
  { \tex_advance:D #1 \__int_eval:w #2 \__int_eval_end: }
\cs_new_protected:Npn \int_sub:Nn #1#2
  { \tex_advance:D #1 - \__int_eval:w #2 \__int_eval_end: }
\cs_new_protected:Npn \int_gadd:Nn #1#2
  { \tex_global:D \tex_advance:D #1 \__int_eval:w #2 \__int_eval_end: }
\cs_new_protected:Npn \int_gsub:Nn #1#2
  { \tex_global:D \tex_advance:D #1 - \__int_eval:w #2 \__int_eval_end: }
\cs_generate_variant:Nn \int_add:Nn  { c }
\cs_generate_variant:Nn \int_gadd:Nn { c }
\cs_generate_variant:Nn \int_sub:Nn  { c }
\cs_generate_variant:Nn \int_gsub:Nn { c }
```

(*End of definition for* `\int_add:Nn` *and others. These functions are documented on page 169.*)

`\int_incr:N` `\int_incr:c` `\int_gincr:N` `\int_gincr:c` `\int_decr:N` `\int_decr:c` `\int_gdecr:N` `\int_gdecr:c`

Incrementing and decrementing of integer registers is done with the following functions.

```
\cs_new_protected:Npn \int_incr:N #1
  { \tex_advance:D #1 \c_one_int }
\cs_new_protected:Npn \int_decr:N #1
  { \tex_advance:D #1 - \c_one_int }
\cs_new_protected:Npn \int_gincr:N #1
  { \tex_global:D \tex_advance:D #1 \c_one_int }
```

```
\cs_new_protected:Npn \int_gdecr:N #1
  { \tex_global:D \tex_advance:D #1 - \c_one_int }
\cs_generate_variant:Nn \int_incr:N  { c }
\cs_generate_variant:Nn \int_decr:N  { c }
\cs_generate_variant:Nn \int_gincr:N { c }
\cs_generate_variant:Nn \int_gdecr:N { c }
```

(*End of definition for* `\int_incr:N` *and others. These functions are documented on page 169.*)

`\int_set:Nn` `\int_set:cn` `\int_set:NV` `\int_set:cV` `\int_gset:Nn` `\int_gset:cn` `\int_gset:NV` `\int_gset:cV`

As integers are register-based TEX issues an error if they are not defined. While the `=` sign is optional, this version with `=` is slightly quicker than without, while adding the optional space after `=` slows things down minutely.

```
\cs_new_protected:Npn \int_set:Nn #1#2
  { #1 = \__int_eval:w #2 \__int_eval_end: }
\cs_new_protected:Npn \int_gset:Nn #1#2
  { \tex_global:D #1 = \__int_eval:w #2 \__int_eval_end: }
\cs_generate_variant:Nn \int_set:Nn  { NV , c , cV }
\cs_generate_variant:Nn \int_gset:Nn { NV , c , cV }
```

(*End of definition for* `\int_set:Nn` *and* `\int_gset:Nn`*. These functions are documented on page 169.*)

59.4 Using integers

`\int_use:N` `\int_use:c`

Here is how counters are accessed. We hand-code the `c` variant for some speed gain.

```
\cs_new_eq:NN \int_use:N \tex_the:D
\cs_new:Npn \int_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(*End of definition for* `\int_use:N`*. This function is documented on page 169.*)

59.5 Integer expression conditionals

`\__int_compare_error:` `\__int_compare_error:Nw`

Those functions are used for comparison tests which use a simple syntax where only one set of braces is required and additional operators such as `!=` and `>=` are supported. The tests first evaluate their left-hand side, with a trailing `\__int_compare_error:`. This marker is normally not expanded, but if the relation symbol is missing from the test's argument, then the marker inserts `=` (and itself) after triggering the relevant TEX error. If the first token which appears after evaluating and removing the left-hand side is not a known relation symbol, then a judiciously placed `\__int_compare_error:Nw` gets expanded, cleaning up the end of the test and telling the user what the problem was.

```
\cs_new_protected:Npn \__int_compare_error:
  {
    \if_int_compare:w \c_zero_int \c_zero_int \fi:
    =
    \__int_compare_error:
  }
\cs_new:Npn \__int_compare_error:Nw
    #1#2 \s__int_stop
  {
    { }
    \c_zero_int \fi:
    \msg_expandable_error:nnn
      { kernel } { unknown-comparison } {#1}
```

```
    \prg_return_false:
  }
```

(*End of definition for* `\__int_compare_error:` *and* `\__int_compare_error:Nw`.)

`\int_compare_p:n`
`\int_compare:nTF`
`\__int_compare:w`
`\__int_compare:Nw`
`\__int_compare:NNw`
`\__int_compare:nnN`
`\__int_compare_end_=:NNw`
`\__int_compare_=:NNw`
`\__int_compare_<:NNw`
`\__int_compare_>:NNw`
`\__int_compare_==:NNw`
`\__int_compare_!=:NNw`
`\__int_compare_<=:NNw`
`\__int_compare_>=:NNw`

Comparison tests using a simple syntax where only one set of braces is required, additional operators such as `!=` and `>=` are supported, and multiple comparisons can be performed at once, for instance `0 < 5 <= 1`. The idea is to loop through the argument, finding one operand at a time, and comparing it to the previous one. The looping auxiliary `\__int_compare:Nw` reads one ⟨*operand*⟩ and one ⟨*comparison*⟩ symbol, and leaves roughly

```
⟨operand⟩ \prg_return_false: \fi:
\reverse_if:N \if_int_compare:w ⟨operand⟩ ⟨comparison⟩
\__int_compare:Nw
```

in the input stream. Each call to this auxiliary provides the second operand of the last call's `\if_int_compare:w`. If one of the ⟨*comparisons*⟩ is `false`, the `true` branch of the TeX conditional is taken (because of `\reverse_if:N`), immediately returning `false` as the result of the test. There is no TeX conditional waiting the first operand, so we add an `\if_false:` and expand by hand with `\int_value:w`, thus skipping `\prg_return_false:` on the first iteration.

Before starting the loop, the first step is to make sure that there is at least one relation symbol. We first let TeX evaluate this left hand side of the (in)equality using `\__int_eval:w`. Since the relation symbols `<`, `>`, `=` and `!` are not allowed in integer expressions, they would terminate the expression. If the argument contains no relation symbol, `\__int_compare_error:` is expanded, inserting `=` and itself after an error. In all cases, `\__int_compare:w` receives as its argument an integer, a relation symbol, and some more tokens. We then setup the loop, which is ended by the two odd-looking items `e` and `{=nd_}`, with a trailing `\s__int_stop` used to grab the entire argument when necessary.

```
\prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
  {
    \exp_after:wN \__int_compare:w
    \int_value:w \__int_eval:w #1 \__int_compare_error:
  }
\cs_new:Npn \__int_compare:w #1 \__int_compare_error:
  {
    \exp_after:wN \if_false: \int_value:w
      \__int_compare:Nw #1 e { = nd_ } \s__int_stop
  }
```

The goal here is to find an ⟨*operand*⟩ and a ⟨*comparison*⟩. The ⟨*operand*⟩ is already evaluated, but we cannot yet grab it as an argument. To access the following relation symbol, we remove the number by applying `\__int_to_roman:w`, after making sure that the argument becomes non-positive: its roman numeral representation is then empty. Then probe the first two tokens with `\__int_compare:NNw` to determine the relation symbol, building a control sequence from it (`\token_to_str:N` gives better errors if `#1` is not a character). All the extended forms have an extra `=` hence the test for that as a second token. If the relation symbol is unknown, then the control sequence is turned by TeX into `\scan_stop:`, ignored thanks to `\unexpanded`, and `\__int_compare_error:Nw` raises an error.

```
\cs_new:Npn \__int_compare:Nw #1#2 \s__int_stop
  {
```

```
    \exp_after:wN \__int_compare:NNw
      \__int_to_roman:w - 0 #2 \s__int_mark
    #1#2 \s__int_stop
  }
\cs_new:Npn \__int_compare:NNw #1#2#3 \s__int_mark
  {
    \__kernel_exp_not:w
    \use:c
      {
        __int_compare_ \token_to_str:N #1
        \if_meaning:w = #2 =  \fi:
        :NNw
      }
      \__int_compare_error:Nw #1
  }
```

When the last ⟨*operand*⟩ is seen, `\__int_compare:NNw` receives `e` and `=nd_` as arguments, hence calling `\__int_compare_end_=:NNw` to end the loop: return the result of the last comparison (involving the operand that we just found). When a normal relation is found, the appropriate auxiliary calls `\__int_compare:nnN` where `#1` is `\if_int_compare:w` or `\reverse_if:N \if_int_compare:w`, `#2` is the ⟨*operand*⟩, and `#3` is one of `<`, `=`, or `>`. As announced earlier, we leave the ⟨*operand*⟩ for the previous conditional. If this conditional is true the result of the test is known, so we remove all tokens and return `false`. Otherwise, we apply the conditional `#1` to the ⟨*operand*⟩ `#2` and the comparison `#3`, and call `\__int_compare:Nw` to look for additional operands, after evaluating the following expression.

```
\cs_new:cpn { __int_compare_end_=:NNw } #1#2#3 e #4 \s__int_stop
  {
    {#3} \exp_stop_f:
    \prg_return_false: \else: \prg_return_true: \fi:
  }
\cs_new:Npn \__int_compare:nnN #1#2#3
  {
        {#2} \exp_stop_f:
      \prg_return_false: \exp_after:wN \__int_use_none_delimit_by_s_stop:w
    \fi:
    #1 #2 #3 \exp_after:wN \__int_compare:Nw \int_value:w \__int_eval:w
  }
```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding `\__int_compare_error:Nw` ⟨*token*⟩ responsible for error detection.

```
\cs_new:cpn { __int_compare_=:NNw } #1#2#3 =
  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
\cs_new:cpn { __int_compare_<:NNw } #1#2#3 <
  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
\cs_new:cpn { __int_compare_>:NNw } #1#2#3 >
  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }
\cs_new:cpn { __int_compare_==:NNw } #1#2#3 ==
  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
\cs_new:cpn { __int_compare_!=:NNw } #1#2#3 !=
  { \__int_compare:nnN { \if_int_compare:w } {#3} = }
\cs_new:cpn { __int_compare_<=:NNw } #1#2#3 <=
  { \__int_compare:nnN { \if_int_compare:w } {#3} > }
```

```
\cs_new:cpn { __int_compare_>=:NNw } #1#2#3 >=
  { \__int_compare:nnN { \if_int_compare:w } {#3} < }
```

(*End of definition for* `\int_compare:nTF` *and others. This function is documented on page 170.*)

`\int_compare_p:nNn`
`\int_compare:nNnTF`

More efficient but less natural in typing.

```
\prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
  {
    \if_int_compare:w \__int_eval:w #1 #2 \__int_eval:w #3 \__int_eval_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\int_compare:nNnTF`. *This function is documented on page 170.*)

`\int_if_zero_p:n`
`\int_if_zero:nTF`

```
\prg_new_conditional:Npnn \int_if_zero:n #1 { p , T , F , TF }
  {
    \if_int_compare:w \__int_eval:w #1 = \c_zero_int
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\int_if_zero:nTF`. *This function is documented on page 171.*)

`\int_case:nn`
`\int_case:nnTF`
`\__int_case:nnTF`
`\__int_case:nw`
`\__int_case_end:nw`

For integer cases, the first task to fully expand the check condition. The over all idea is then much the same as for `\str_case:nnTF` as described in l3str.

```
\cs_new:Npn \int_case:nnTF #1
  {
    \exp:w
    \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
  }
\cs_new:Npn \int_case:nnT #1#2#3
  {
    \exp:w
    \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
  }
\cs_new:Npn \int_case:nnF #1#2
  {
    \exp:w
    \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
  }
\cs_new:Npn \int_case:nn #1#2
  {
    \exp:w
    \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
  }
\cs_new:Npn \__int_case:nnTF #1#2#3#4
  { \__int_case:nw {#1} #2 {#1} { } \s__int_mark {#3} \s__int_mark {#4} \s__int_stop }
\cs_new:Npn \__int_case:nw #1#2#3
  {
```

```
    \int_compare:nNnTF {#1} = {#2}
      { \__int_case_end:nw {#3} }
      { \__int_case:nw {#1} }
  }
\cs_new:Npn \__int_case_end:nw #1#2#3 \s__int_mark #4#5 \s__int_stop
  { \exp_end: #1 #4 }
```

(*End of definition for* `\int_case:nnTF` *and others. This function is documented on page 171.*)

`\int_if_odd_p:n`
`\int_if_odd:nTF`
`\int_if_even_p:n`
`\int_if_even:nTF`

A predicate function.

```
\prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
  {
    \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
  {
    \reverse_if:N \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\int_if_odd:nTF` *and* `\int_if_even:nTF`*. These functions are documented on page 171.*)

59.6 Integer expression loops

`\int_while_do:nn`
`\int_until_do:nn`
`\int_do_while:nn`
`\int_do_until:nn`

These are quite easy given the above functions. The `while` versions test first and then execute the body. The `do_while` does it the other way round.

```
\cs_new:Npn \int_while_do:nn #1#2
  {
    \int_compare:nT {#1}
      {
        #2
        \int_while_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \int_until_do:nn #1#2
  {
    \int_compare:nF {#1}
      {
        #2
        \int_until_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \int_do_while:nn #1#2
  {
    #2
```

```
    \int_compare:nT {#1}
      { \int_do_while:nn {#1} {#2} }
  }
\cs_new:Npn \int_do_until:nn #1#2
  {
    #2
    \int_compare:nF {#1}
      { \int_do_until:nn {#1} {#2} }
  }
```

(*End of definition for* `\int_while_do:nn` *and others. These functions are documented on page 172.*)

`\int_while_do:nNnn`
`\int_until_do:nNnn`
`\int_do_while:nNnn`
`\int_do_until:nNnn`

As above but not using the more natural syntax.

```
\cs_new:Npn \int_while_do:nNnn #1#2#3#4
  {
    \int_compare:nNnT {#1} #2 {#3}
      {
        #4
        \int_while_do:nNnn {#1} #2 {#3} {#4}
      }
  }
\cs_new:Npn \int_until_do:nNnn #1#2#3#4
  {
    \int_compare:nNnF {#1} #2 {#3}
      {
        #4
        \int_until_do:nNnn {#1} #2 {#3} {#4}
      }
  }
\cs_new:Npn \int_do_while:nNnn #1#2#3#4
  {
    #4
    \int_compare:nNnT {#1} #2 {#3}
      { \int_do_while:nNnn {#1} #2 {#3} {#4} }
  }
\cs_new:Npn \int_do_until:nNnn #1#2#3#4
  {
    #4
    \int_compare:nNnF {#1} #2 {#3}
      { \int_do_until:nNnn {#1} #2 {#3} {#4} }
  }
```

(*End of definition for* `\int_while_do:nNnn` *and others. These functions are documented on page 172.*)

59.7 Integer step functions

`\int_step_function:nnnN`
`\__int_step:wwwN`
`\__int_step:NwnnN`
`\int_step_function:nN`
`\int_step_function:nnN`

Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```
\cs_new:Npn \int_step_function:nnnN #1#2#3
  {
```

```
    \exp_after:wN \__int_step:wwwN
    \int_value:w \__int_eval:w #1 \exp_after:wN ;
    \int_value:w \__int_eval:w #2 \exp_after:wN ;
    \int_value:w \__int_eval:w #3 ;
  }
\cs_new:Npn \__int_step:wwwN #1; #2; #3; #4
  {
    \int_compare:nNnTF {#2} > \c_zero_int
      { \__int_step:NwnnN > }
      {
        \int_compare:nNnTF {#2} = \c_zero_int
          {
            \msg_expandable_error:nnn
              { kernel } { zero-step } {#4}
            \prg_break:
          }
          { \__int_step:NwnnN < }
      }
      #1 ; {#2} {#3} #4
    \prg_break_point:
  }
\cs_new:Npn \__int_step:NwnnN #1#2 ; #3#4#5
  {
    \if_int_compare:w #2 #1 #4 \exp_stop_f:
      \prg_break:n
    \fi:
    #5 {#2}
    \exp_after:wN \__int_step:NwnnN
    \exp_after:wN #1
    \int_value:w \__int_eval:w #2 + #3 ; {#3} {#4} #5
  }
\cs_new:Npn \int_step_function:nN
  { \int_step_function:nnnN { 1 } { 1 } }
\cs_new:Npn \int_step_function:nnN #1
  { \int_step_function:nnnN {#1} { 1 } }
```

(*End of definition for* `\int_step_function:nnnN` *and others. These functions are documented on page 173.*)

`\int_step_inline:nn`
`\int_step_inline:nnn`
`\int_step_inline:nnnn`
`\int_step_variable:nNn`
`\int_step_variable:nnNn`
`\int_step_variable:nnnNn`
`\__int_step:NNnnnn`

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\int_step_function:nnnN`. We put a `\prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g__kernel_prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so that no breaking function recognizes this break point as its own.

```
\cs_new_protected:Npn \int_step_inline:nn
  { \int_step_inline:nnnn { 1 } { 1 } }
\cs_new_protected:Npn \int_step_inline:nnn #1
  { \int_step_inline:nnnn {#1} { 1 } }
\cs_new_protected:Npn \int_step_inline:nnnn
  {
    \int_gincr:N \g__kernel_prg_map_int
    \exp_args:NNc \__int_step:NNnnnn
      \cs_gset_protected:Npn
```

```
      { __int_map_ \int_use:N \g__kernel_prg_map_int :w }
  }
\cs_new_protected:Npn \int_step_variable:nNn
  { \int_step_variable:nnnNn { 1 } { 1 } }
\cs_new_protected:Npn \int_step_variable:nnNn #1
  { \int_step_variable:nnnNn {#1} { 1 } }
\cs_new_protected:Npn \int_step_variable:nnnNn #1#2#3#4#5
  {
    \int_gincr:N \g__kernel_prg_map_int
    \exp_args:NNc \__int_step:NNnnnn
      \cs_gset_protected:Npe
      { __int_map_ \int_use:N \g__kernel_prg_map_int :w }
      {#1}{#2}{#3}
      {
        \tl_set:Nn \exp_not:N #4 {##1}
        \exp_not:n {#5}
      }
  }
\cs_new_protected:Npn \__int_step:NNnnnn #1#2#3#4#5#6
  {
    #1 #2 ##1 {#6}
    \int_step_function:nnnN {#3} {#4} {#5} #2
    \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
  }
```

(*End of definition for* `\int_step_inline:nn` *and others. These functions are documented on page 173.*)

59.8 Formatting integers

`\int_to_arabic:n`
`\int_to_arabic:v`

Nothing exciting here.

```
\cs_new_eq:NN \int_to_arabic:n \int_eval:n
\cs_generate_variant:Nn \int_to_arabic:n { v }
```

(*End of definition for* `\int_to_arabic:n`*. This function is documented on page 174.*)

`\int_to_symbols:nnn`
`\__int_to_symbols:nnnn`
`\__int_to_symbols:ennn`

For conversion of integers to arbitrary symbols the method is in general as follows. The input number (`#1`) is compared to the total number of symbols available at each place (`#2`). If the input is larger than the total number of symbols available then the modulus is needed, with one added so that the positions don't have to number from zero. Using an `f`-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a 'nice' number at each stage. Of course, if the initial input was small enough then there is no problem and everything is easy.

```
\cs_new:Npn \int_to_symbols:nnn #1#2#3
  {
    \int_compare:nNnTF {#1} > {#2}
      {
        \__int_to_symbols:ennn
          {
            \int_case:nn
              { 1 + \int_mod:nn { #1 - 1 } {#2} }
              {#3}
          }
          {#1} {#2} {#3}
```

```
    }
    { \int_case:nn {#1} {#3} }
  }
\cs_new:Npn \__int_to_symbols:nnnn #1#2#3#4
  {
    \exp_args:Nf \int_to_symbols:nnn
      { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}
    #1
  }
\cs_generate_variant:Nn \__int_to_symbols:nnnn { e }
```

(*End of definition for* `\int_to_symbols:nnn` *and* `\__int_to_symbols:nnnn`. *This function is documented on page 174.*)

`\int_to_alph:n`
`\int_to_Alph:n`

These both use the above function with input functions that make sense for the alphabet in English.

```
\cs_new:Npn \int_to_alph:n #1
  {
    \int_to_symbols:nnn {#1} { 26 }
      {
        {  1 } { a }
        {  2 } { b }
        {  3 } { c }
        {  4 } { d }
        {  5 } { e }
        {  6 } { f }
        {  7 } { g }
        {  8 } { h }
        {  9 } { i }
        { 10 } { j }
        { 11 } { k }
        { 12 } { l }
        { 13 } { m }
        { 14 } { n }
        { 15 } { o }
        { 16 } { p }
        { 17 } { q }
        { 18 } { r }
        { 19 } { s }
        { 20 } { t }
        { 21 } { u }
        { 22 } { v }
        { 23 } { w }
        { 24 } { x }
        { 25 } { y }
        { 26 } { z }
      }
  }
\cs_new:Npn \int_to_Alph:n #1
  {
    \int_to_symbols:nnn {#1} { 26 }
      {
        {  1 } { A }
        {  2 } { B }
```

```
        {  3 } { C }
        {  4 } { D }
        {  5 } { E }
        {  6 } { F }
        {  7 } { G }
        {  8 } { H }
        {  9 } { I }
        { 10 } { J }
        { 11 } { K }
        { 12 } { L }
        { 13 } { M }
        { 14 } { N }
        { 15 } { O }
        { 16 } { P }
        { 17 } { Q }
        { 18 } { R }
        { 19 } { S }
        { 20 } { T }
        { 21 } { U }
        { 22 } { V }
        { 23 } { W }
        { 24 } { X }
        { 25 } { Y }
        { 26 } { Z }
      }
  }
```

(End of definition for `\int_to_alph:n` and `\int_to_Alph:n`. These functions are documented on page 174.)

`\int_to_base:nn`
`\int_to_Base:nn`
`\__int_to_base:nn`
`\__int_to_Base:nn`
`\__int_to_base:nnN`
`\__int_to_Base:nnN`
`\__int_to_base:nnnN`
`\__int_to_Base:nnnN`
`\__int_to_letter:n`
`\__int_to_Letter:n`

Converting from base ten (`#1`) to a second base (`#2`) starts with computing `#1`: if it is a complicated calculation, we shouldn't perform it twice. Then check the sign, store it, either `-` or `\c_empty_tl`, and feed the absolute value to the next auxiliary function.

```
\cs_new:Npn \int_to_base:nn #1
  { \exp_args:Nf \__int_to_base:nn { \int_eval:n {#1} } }
\cs_new:Npn \int_to_Base:nn #1
  { \exp_args:Nf \__int_to_Base:nn { \int_eval:n {#1} } }
\cs_new:Npn \__int_to_base:nn #1#2
  {
    \int_compare:nNnTF {#1} < 0
      { \exp_args:No \__int_to_base:nnN { \use_none:n #1 } {#2} - }
      { \__int_to_base:nnN {#1} {#2} \c_empty_tl }
  }
\cs_new:Npn \__int_to_Base:nn #1#2
  {
    \int_compare:nNnTF {#1} < 0
      { \exp_args:No \__int_to_Base:nnN { \use_none:n #1 } {#2} - }
      { \__int_to_Base:nnN {#1} {#2} \c_empty_tl }
  }
```

Here, the idea is to provide a recursive system to deal with the input. The output is built up after the end of the function. At each pass, the value in `#1` is checked to see if it is less than the new base (`#2`). If it is, then it is converted directly, putting the sign back in front. On the other hand, if the value to convert is greater than or equal to the new base then the modulus and remainder values are found. The modulus is converted to a symbol and put on the right, and the remainder is carried forward to the next round.

```
\cs_new:Npn \__int_to_base:nnN #1#2#3
  {
    \int_compare:nNnTF {#1} < {#2}
      { \exp_last_unbraced:Nf #3 { \__int_to_letter:n {#1} } }
      {
        \exp_args:Nf \__int_to_base:nnnN
          { \__int_to_letter:n { \int_mod:nn {#1} {#2} } }
          {#1}
          {#2}
          #3
      }
  }
\cs_new:Npn \__int_to_base:nnnN #1#2#3#4
  {
    \exp_args:Nf \__int_to_base:nnN
      { \int_div_truncate:nn {#2} {#3} }
      {#3}
      #4
    #1
  }
\cs_new:Npn \__int_to_Base:nnN #1#2#3
  {
    \int_compare:nNnTF {#1} < {#2}
      { \exp_last_unbraced:Nf #3 { \__int_to_Letter:n {#1} } }
      {
        \exp_args:Nf \__int_to_Base:nnnN
          { \__int_to_Letter:n { \int_mod:nn {#1} {#2} } }
          {#1}
          {#2}
          #3
      }
  }
\cs_new:Npn \__int_to_Base:nnnN #1#2#3#4
  {
    \exp_args:Nf \__int_to_Base:nnN
      { \int_div_truncate:nn {#2} {#3} }
      {#3}
      #4
    #1
  }
```

Convert to a letter only if necessary, otherwise simply return the value unchanged. It would be cleaner to use `\int_case:nn`, but in our case, the cases are contiguous, so it is forty times faster to use the `\if_case:w` primitive. The first `\exp_after:wN` expands the conditional, jumping to the correct case, the second one expands after the resulting character to close the conditional. Since `#1` might be an expression, and not directly a single digit, we need to evaluate it properly, and expand the trailing `\fi:`.

```
\cs_new:Npn \__int_to_letter:n #1
  {
    \exp_after:wN \exp_after:wN
    \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
         a
```

```
    \or: b
    \or: c
    \or: d
    \or: e
    \or: f
    \or: g
    \or: h
    \or: i
    \or: j
    \or: k
    \or: l
    \or: m
    \or: n
    \or: o
    \or: p
    \or: q
    \or: r
    \or: s
    \or: t
    \or: u
    \or: v
    \or: w
    \or: x
    \or: y
    \or: z
    \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
    \fi:
  }
\cs_new:Npn \__int_to_Letter:n #1
  {
    \exp_after:wN \exp_after:wN
    \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
         A
    \or: B
    \or: C
    \or: D
    \or: E
    \or: F
    \or: G
    \or: H
    \or: I
    \or: J
    \or: K
    \or: L
    \or: M
    \or: N
    \or: O
    \or: P
    \or: Q
    \or: R
    \or: S
    \or: T
    \or: U
    \or: V
```

```
    \or: W
    \or: X
    \or: Y
    \or: Z
    \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
    \fi:
  }
```

(*End of definition for* `\int_to_base:nn` *and others. These functions are documented on page* 175.)

`\int_to_bin:n`
`\int_to_hex:n`
`\int_to_Hex:n`
`\int_to_oct:n`

Wrappers around the generic function.

```
\cs_new:Npn \int_to_bin:n #1
  { \int_to_base:nn {#1} { 2 } }
\cs_new:Npn \int_to_hex:n #1
  { \int_to_base:nn {#1} { 16 } }
\cs_new:Npn \int_to_Hex:n #1
  { \int_to_Base:nn {#1} { 16 } }
\cs_new:Npn \int_to_oct:n #1
  { \int_to_base:nn {#1} { 8 } }
```

(*End of definition for* `\int_to_bin:n` *and others. These functions are documented on page* 175.)

`\int_to_roman:n`
`\int_to_Roman:n`
`\__int_to_roman:N`
`\__int_to_roman:N`
`\__int_to_roman_i:w`
`\__int_to_roman_v:w`
`\__int_to_roman_x:w`
`\__int_to_roman_l:w`
`\__int_to_roman_c:w`
`\__int_to_roman_d:w`
`\__int_to_roman_m:w`
`\__int_to_roman_Q:w`
`\__int_to_Roman_i:w`
`\__int_to_Roman_v:w`
`\__int_to_Roman_x:w`
`\__int_to_Roman_l:w`
`\__int_to_Roman_c:w`
`\__int_to_Roman_d:w`
`\__int_to_Roman_m:w`
`\__int_to_Roman_Q:w`

The `\__int_to_roman:w` primitive creates tokens of category code 12 (other). Usually, what is actually wanted is letters. The approach here is to convert the output of the primitive into letters using appropriate control sequence names. That keeps everything expandable. The loop is terminated by the conversion of the `Q`.

```
\cs_new:Npn \int_to_roman:n #1
  {
    \exp_after:wN \__int_to_roman:N
      \__int_to_roman:w \int_eval:n {#1} Q
  }
\cs_new:Npn \__int_to_roman:N #1
  {
    \use:c { __int_to_roman_ #1 :w }
    \__int_to_roman:N
  }
\cs_new:Npn \int_to_Roman:n #1
  {
    \exp_after:wN \__int_to_Roman_aux:N
      \__int_to_roman:w \int_eval:n {#1} Q
  }
\cs_new:Npn \__int_to_Roman_aux:N #1
  {
    \use:c { __int_to_Roman_ #1 :w }
    \__int_to_Roman_aux:N
  }
\cs_new:Npn \__int_to_roman_i:w { i }
\cs_new:Npn \__int_to_roman_v:w { v }
\cs_new:Npn \__int_to_roman_x:w { x }
\cs_new:Npn \__int_to_roman_l:w { l }
\cs_new:Npn \__int_to_roman_c:w { c }
\cs_new:Npn \__int_to_roman_d:w { d }
\cs_new:Npn \__int_to_roman_m:w { m }
\cs_new:Npn \__int_to_roman_Q:w #1 { }
```

```
\cs_new:Npn \__int_to_Roman_i:w { I }
\cs_new:Npn \__int_to_Roman_v:w { V }
\cs_new:Npn \__int_to_Roman_x:w { X }
\cs_new:Npn \__int_to_Roman_l:w { L }
\cs_new:Npn \__int_to_Roman_c:w { C }
\cs_new:Npn \__int_to_Roman_d:w { D }
\cs_new:Npn \__int_to_Roman_m:w { M }
\cs_new:Npn \__int_to_Roman_Q:w #1 { }
```

(*End of definition for* `\int_to_roman:n` *and others. These functions are documented on page 175.*)

59.9 Converting from other formats to integers

`\__int_pass_signs:wn`
`\__int_pass_signs_end:wn`

Called as `\__int_pass_signs:wn` ⟨*signs and digits*⟩ `\s__int_stop {`⟨*code*⟩`}`, this function leaves in the input stream any sign it finds, then inserts the ⟨*code*⟩ before the first non-sign token (and removes `\s__int_stop`). More precisely, it deletes any `+` and passes any `-` to the input stream, hence should be called in an integer expression.

```
\cs_new:Npn \__int_pass_signs:wn #1
  {
    \if:w + \if:w - \exp_not:N #1 + \fi: \exp_not:N #1
      \exp_after:wN \__int_pass_signs:wn
    \else:
      \exp_after:wN \__int_pass_signs_end:wn
      \exp_after:wN #1
    \fi:
  }
\cs_new:Npn \__int_pass_signs_end:wn #1 \s__int_stop #2 { #2 #1 }
```

(*End of definition for* `\__int_pass_signs:wn` *and* `\__int_pass_signs_end:wn`*.*)

`\int_from_alph:n`
`\__int_from_alph:nN`
`\__int_from_alph:N`

First take care of signs then loop through the input using the `recursion` quarks. The `\__int_from_alph:nN` auxiliary collects in its first argument the value obtained so far, and the auxiliary `\__int_from_alph:N` converts one letter to an expression which evaluates to the correct number.

```
\cs_new:Npn \int_from_alph:n #1
  {
    \int_eval:n
      {
        \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
          \s__int_stop { \__int_from_alph:nN { 0 } }
        \q__int_recursion_tail \q__int_recursion_stop
      }
  }
\cs_new:Npn \__int_from_alph:nN #1#2
  {
    \__int_if_recursion_tail_stop_do:Nn #2 {#1}
    \exp_args:Nf \__int_from_alph:nN
      { \int_eval:n { #1 * 26 + \__int_from_alph:N #2 } }
  }
\cs_new:Npn \__int_from_alph:N #1
  { `#1 - \int_compare:nNnTF { `#1 } < { 91 } { 64 } { 96 } }
```

(*End of definition for* `\int_from_alph:n`*,* `\__int_from_alph:nN`*, and* `\__int_from_alph:N`*. This function is documented on page 175.*)

\int_from_base:nn
__int_from_base:nnN
__int_from_base:N

Leave the signs into the integer expression, then loop through characters, collecting the value found so far in the first argument of `\__int_from_base:nnN`. To convert a single character, `\__int_from_base:N` checks first for digits, then distinguishes lower from upper case letters, turning them into the appropriate number. Note that this auxiliary does not use `\int_eval:n`, hence is not safe for general use.

```
\cs_new:Npn \int_from_base:nn #1#2
  {
    \int_eval:n
      {
        \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
          \s__int_stop { \__int_from_base:nnN { 0 } {#2} }
        \q__int_recursion_tail \q__int_recursion_stop
      }
  }
\cs_new:Npn \__int_from_base:nnN #1#2#3
  {
    \__int_if_recursion_tail_stop_do:Nn #3 {#1}
    \exp_args:Nf \__int_from_base:nnN
      { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
      {#2}
  }
\cs_new:Npn \__int_from_base:N #1
  {
    \int_compare:nNnTF { `#1 } < { 58 }
      {#1}
      { `#1 - \int_compare:nNnTF { `#1 } < { 91 } { 55 } { 87 } }
  }
```

(*End of definition for* `\int_from_base:nn`*,* `\__int_from_base:nnN`*, and* `\__int_from_base:N`*. This function is documented on page 176.*)

\int_from_bin:n
\int_from_hex:n
\int_from_oct:n

Wrappers around the generic function.

```
\cs_new:Npn \int_from_bin:n #1
  { \int_from_base:nn {#1} { 2 } }
\cs_new:Npn \int_from_hex:n #1
  { \int_from_base:nn {#1} { 16 } }
\cs_new:Npn \int_from_oct:n #1
  { \int_from_base:nn {#1} { 8 } }
```

(*End of definition for* `\int_from_bin:n`*,* `\int_from_hex:n`*, and* `\int_from_oct:n`*. These functions are documented on page 175.*)

\c__int_from_roman_i_int
\c__int_from_roman_v_int
\c__int_from_roman_x_int
\c__int_from_roman_l_int
\c__int_from_roman_c_int
\c__int_from_roman_d_int
\c__int_from_roman_m_int
\c__int_from_roman_I_int
\c__int_from_roman_V_int
\c__int_from_roman_X_int
\c__int_from_roman_L_int
\c__int_from_roman_C_int
\c__int_from_roman_D_int
\c__int_from_roman_M_int

Constants used to convert from Roman numerals to integers.

```
\int_const:cn { c__int_from_roman_i_int } { 1 }
\int_const:cn { c__int_from_roman_v_int } { 5 }
\int_const:cn { c__int_from_roman_x_int } { 10 }
\int_const:cn { c__int_from_roman_l_int } { 50 }
\int_const:cn { c__int_from_roman_c_int } { 100 }
\int_const:cn { c__int_from_roman_d_int } { 500 }
\int_const:cn { c__int_from_roman_m_int } { 1000 }
\int_const:cn { c__int_from_roman_I_int } { 1 }
\int_const:cn { c__int_from_roman_V_int } { 5 }
\int_const:cn { c__int_from_roman_X_int } { 10 }
\int_const:cn { c__int_from_roman_L_int } { 50 }
```

```
\int_const:cn { c__int_from_roman_C_int } { 100 }
\int_const:cn { c__int_from_roman_D_int } { 500 }
\int_const:cn { c__int_from_roman_M_int } { 1000 }
```

(*End of definition for* `\c__int_from_roman_i_int` *and others.*)

`\int_from_roman:n`
`\__int_from_roman:NN`
`\__int_from_roman_error:w`

The method here is to iterate through the input, finding the appropriate value for each letter and building up a sum. This is then evaluated by TeX. If any unknown letter is found, skip to the closing parenthesis and insert `*0-1` afterwards, to replace the value by −1.

```
\cs_new:Npn \int_from_roman:n #1
  {
    \int_eval:n
      {
        (
          0
          \exp_after:wN \__int_from_roman:NN \tl_to_str:n {#1}
          \q__int_recursion_tail \q__int_recursion_tail \q__int_recursion_stop
        )
      }
  }
\cs_new:Npn \__int_from_roman:NN #1#2
  {
    \__int_if_recursion_tail_stop:N #1
    \int_if_exist:cF { c__int_from_roman_ #1 _int }
      { \__int_from_roman_error:w }
    \__int_if_recursion_tail_stop_do:Nn #2
      { + \use:c { c__int_from_roman_ #1 _int } }
    \int_if_exist:cF { c__int_from_roman_ #2 _int }
      { \__int_from_roman_error:w }
    \int_compare:nNnTF
      { \use:c { c__int_from_roman_ #1 _int } }
      <
      { \use:c { c__int_from_roman_ #2 _int } }
      {
        + \use:c { c__int_from_roman_ #2 _int }
        - \use:c { c__int_from_roman_ #1 _int }
        \__int_from_roman:NN
      }
      {
        + \use:c { c__int_from_roman_ #1 _int }
        \__int_from_roman:NN #2
      }
  }
\cs_new:Npn \__int_from_roman_error:w #1 \q__int_recursion_stop #2
  { #2 * 0 - 1 }
```

(*End of definition for* `\int_from_roman:n`*,* `\__int_from_roman:NN`*, and* `\__int_from_roman_error:w`*. This function is documented on page 176.*)

59.10 Viewing integer

`\int_show:N`
`\int_show:c`
`\__int_show:nN`

Diagnostics.

```
\cs_new_eq:NN \int_show:N \__kernel_register_show:N
\cs_generate_variant:Nn \int_show:N { c }
```

(*End of definition for* `\int_show:N` *and* `\__int_show:nN`*. This function is documented on page 176.*)

`\int_show:n` We don't use the TeX primitive `\showthe` to show integer expressions: this gives a more unified output.

```
\cs_new_protected:Npn \int_show:n
  { \__kernel_msg_show_eval:Nn \int_eval:n }
```

(*End of definition for* `\int_show:n`*. This function is documented on page 177.*)

`\int_log:N` `\int_log:c` Diagnostics.

```
\cs_new_eq:NN \int_log:N \__kernel_register_log:N
\cs_generate_variant:Nn \int_log:N { c }
```

(*End of definition for* `\int_log:N`*. This function is documented on page 177.*)

`\int_log:n` Similar to `\int_show:n`.

```
\cs_new_protected:Npn \int_log:n
  { \__kernel_msg_log_eval:Nn \int_eval:n }
```

(*End of definition for* `\int_log:n`*. This function is documented on page 177.*)

59.11 Random integers

`\int_rand:nn` Defined in l3fp-random.

(*End of definition for* `\int_rand:nn`*. This function is documented on page 176.*)

59.12 Constant integers

`\c_zero_int` `\c_one_int` The zero is defined in l3basics.

```
\int_const:Nn \c_one_int { 1 }
```

(*End of definition for* `\c_zero_int` *and* `\c_one_int`*. These variables are documented on page 177.*)

`\c_max_int` The largest number allowed is $2^{31}-1$

```
\int_const:Nn \c_max_int { 2 147 483 647 }
```

(*End of definition for* `\c_max_int`*. This variable is documented on page 177.*)

`\c_max_char_int` The largest character code is 1114111 (hexadecimal `10FFFF`) in XeTeX and LuaTeX and 255 in other engines. In many places pTeX and upTeX support larger character codes but for instance the values of `\lccode` are restricted to $[0, 255]$.

```
\int_const:Nn \c_max_char_int
  {
    \if_int_odd:w 0
      \cs_if_exist:NT \tex_luatexversion:D  { 1 }
      \cs_if_exist:NT \tex_XeTeXversion:D   { 1 } ~
      "10FFFF
    \else:
      "FF
    \fi:
  }
```

(*End of definition for* `\c_max_char_int`*. This variable is documented on page 177.*)

59.13 Scratch integers

\l_tmpa_int
\l_tmpb_int
\g_tmpa_int
\g_tmpb_int

We provide two local and two global scratch counters, maybe we need more or less.

```
\int_new:N \l_tmpa_int
\int_new:N \l_tmpb_int
\int_new:N \g_tmpa_int
\int_new:N \g_tmpb_int
```

(*End of definition for* `\l_tmpa_int` *and others. These variables are documented on page 177.*)

59.14 Integers for earlier modules

<@@=seq>

\l__int_internal_a_int
\l__int_internal_b_int

```
\int_new:N \l__int_internal_a_int
\int_new:N \l__int_internal_b_int
```

(*End of definition for* `\l__int_internal_a_int` *and* `\l__int_internal_b_int`*.*)

```
⟨/package⟩
```

Chapter 60

l3flag implementation

```
⟨*package⟩
⟨@@=flag⟩
```

The following test files are used for this code: m3flag001.

60.1 Protected flag commands

The height h of a flag (which is initially zero) is stored by setting control sequences of the form `\`⟨*flag name*⟩⟨*integer*⟩ to `\relax` for $0 \le \langle integer \rangle < h$. These control sequences are produced by `\cs:w` ⟨*flag var*⟩ ⟨*integer*⟩ `\cs_end:`, namely the ⟨*flag var*⟩ is actually a (protected) macro expanding to its own csname.

`\flag_new:N`
`\flag_new:c`

Evaluate the csname of `#1` for use in constructing the various indexed macros.

```
\cs_new_protected:Npn \flag_new:N #1
  { \cs_new_protected:Npe #1 { \cs_to_str:N #1 } }
\cs_generate_variant:Nn \flag_new:N { c }
```

(*End of definition for* `\flag_new:N`. *This function is documented on page 180.*)

`\l_tmpa_flag`
`\l_tmpb_flag`

Two flag variables for scratch use.

```
\flag_new:N \l_tmpa_flag
\flag_new:N \l_tmpb_flag
```

(*End of definition for* `\l_tmpa_flag` *and* `\l_tmpb_flag`. *These variables are documented on page 182.*)

`\flag_clear:N`
`\flag_clear:c`
`\__flag_clear:wN`

Undefine control sequences, starting from the `0` flag, upwards, until reaching an undefined control sequence. We don't use `\cs_undefine:c` because that would act globally.

```
\cs_new_protected:Npn \flag_clear:N #1
  {
    \__flag_clear:wN 0 ; #1
    \prg_break_point:
  }
\cs_generate_variant:Nn \flag_clear:N { c }
\cs_new_protected:Npn \__flag_clear:wN #1 ; #2
  {
    \if_cs_exist:w #2 #1 \cs_end: \else:
      \prg_break:n
    \fi:
```

```
    \cs_set_eq:cN { #2 #1 } \tex_undefined:D
    \exp_after:wN \__flag_clear:wN
    \int_value:w \int_eval:w \c_one_int + #1 ; #2
  }
```

(*End of definition for* `\flag_clear:N` *and* `\__flag_clear:wN`*. This function is documented on page 181.*)

`\flag_clear_new:N`
`\flag_clear_new:c`

As for other datatypes, clear the ⟨*flag var*⟩ or create a new one, as appropriate.

```
\cs_new_protected:Npn \flag_clear_new:N #1
  { \flag_if_exist:NTF #1 { \flag_clear:N } { \flag_new:N } #1 }
\cs_generate_variant:Nn \flag_clear_new:N { c }
```

(*End of definition for* `\flag_clear_new:N`*. This function is documented on page 181.*)

`\flag_show:N`
`\flag_show:c`
`\flag_log:N`
`\flag_log:c`
`\__flag_show:NN`

Show the height (terminal or log file) using appropriate l3msg auxiliaries.

```
\cs_new_protected:Npn \flag_show:N { \__flag_show:NN \tl_show:n }
\cs_generate_variant:Nn \flag_show:N { c }
\cs_new_protected:Npn \flag_log:N { \__flag_show:NN \tl_log:n }
\cs_generate_variant:Nn \flag_log:N { c }
\cs_new_protected:Npn \__flag_show:NN #1#2
  {
    \__kernel_chk_defined:NT #2
      { \exp_args:Ne #1 { \tl_to_str:n { #2 height } = \flag_height:N #2 } }
  }
```

(*End of definition for* `\flag_show:N`*,* `\flag_log:N`*, and* `\__flag_show:NN`*. These functions are documented on page 181.*)

60.2 Expandable flag commands

`\flag_if_exist_p:N`
`\flag_if_exist_p:c`
`\flag_if_exist:NTF`
`\flag_if_exist:cTF`

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \flag_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \flag_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End of definition for* `\flag_if_exist:NTF`*. This function is documented on page 181.*)

`\flag_if_raised_p:N`
`\flag_if_raised_p:c`
`\flag_if_raised:NTF`
`\flag_if_raised:cTF`

Test if the flag has a non-zero height, by checking the `0` control sequence.

```
\prg_new_conditional:Npnn \flag_if_raised:N #1 { p , T , F , TF }
  {
    \if_cs_exist:w #1 0 \cs_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_generate_conditional_variant:Nnn \flag_if_raised:N
  { c } { p , T , F , TF }
```

(*End of definition for* `\flag_if_raised:NTF`*. This function is documented on page 181.*)

\flag_height:N
\flag_height:c
__flag_height_loop:wN
__flag_height_end:wN

Extract the value of the flag by going through all of the control sequences starting from 0.

```
\cs_new:Npn \flag_height:N #1 { \__flag_height_loop:wN 0; #1 }
\cs_new:Npn \__flag_height_loop:wN #1 ; #2
  {
    \if_cs_exist:w #2 #1 \cs_end: \else:
      \exp_after:wN \__flag_height_end:wN
    \fi:
    \exp_after:wN \__flag_height_loop:wN
    \int_value:w \int_eval:w \c_one_int + #1 ; #2
  }
\cs_new:Npn \__flag_height_end:wN #1 + #2 ; #3 {#2}
\cs_generate_variant:Nn \flag_height:N { c }
```

(*End of definition for* `\flag_height:N`, `\__flag_height_loop:wN`, *and* `\__flag_height_end:wN`. *This function is documented on page 181.*)

\flag_raise:N
\flag_raise:c

Change the appropriate control sequence to `\relax` by expanding a `\cs:w` … `\cs_end:` construction, then pass it to `\use_none:n` to avoid leaving anything in the input stream.

```
\cs_new:Npn \flag_raise:N #1
  { \exp_after:wN \use_none:n \cs:w #1 \flag_height:N #1 \cs_end: }
\cs_generate_variant:Nn \flag_raise:N { c }
```

(*End of definition for* `\flag_raise:N`. *This function is documented on page 181.*)

\flag_ensure_raised:N
\flag_ensure_raised:c

Pass the control sequence with name ⟨*flag name*⟩0 to `\use_none:n`. Constructing the control sequence ensures that it changes from being undefined (if it was so) to being `\relax`.

```
\cs_new:Npn \flag_ensure_raised:N #1
  { \exp_after:wN \use_none:n \cs:w #1 0 \cs_end: }
\cs_generate_variant:Nn \flag_ensure_raised:N { c }
```

(*End of definition for* `\flag_ensure_raised:N`. *This function is documented on page 181.*)

60.3 Old n-type flag commands

Here we keep the old flag commands since our policy is to no longer delete deprecated functions. The idea is to simply map ⟨*flag name*⟩ to `\l_`⟨*flag name*⟩`_flag`. When the debugging code is activated, it checks existence of the N-type flag variables that result.

\flag_new:n
\flag_clear:n
\flag_clear_new:n
\flag_if_exist_p:n
\flag_if_exist:nTF
\flag_if_raised_p:n
\flag_if_raised:nTF
\flag_height:n
\flag_raise:n
\flag_ensure_raised:n

```
\cs_new_protected:Npn \flag_new:n #1 { \flag_new:c { l_#1_flag } }
\cs_new_protected:Npn \flag_clear:n #1 { \flag_clear:c { l_#1_flag } }
\cs_new_protected:Npn \flag_clear_new:n #1 { \flag_clear_new:c { l_#1_flag } }
\cs_new:Npn \flag_if_exist_p:n #1 { \flag_if_exist_p:c { l_#1_flag } }
\cs_new:Npn \flag_if_exist:nT #1 { \flag_if_exist:cT { l_#1_flag } }
\cs_new:Npn \flag_if_exist:nF #1 { \flag_if_exist:cF { l_#1_flag } }
\cs_new:Npn \flag_if_exist:nTF #1 { \flag_if_exist:cTF { l_#1_flag } }
\cs_new:Npn \flag_if_raised_p:n #1 { \flag_if_raised_p:c { l_#1_flag } }
\cs_new:Npn \flag_if_raised:nT #1 { \flag_if_raised:cT { l_#1_flag } }
\cs_new:Npn \flag_if_raised:nF #1 { \flag_if_raised:cF { l_#1_flag } }
\cs_new:Npn \flag_if_raised:nTF #1 { \flag_if_raised:cTF { l_#1_flag } }
\cs_new:Npn \flag_height:n #1 { \flag_height:c { l_#1_flag } }
```

```
\cs_new:Npn \flag_raise:n #1 { \flag_raise:c { l_#1_flag } }
\cs_new:Npn \flag_ensure_raised:n #1 { \flag_ensure_raised:c { l_#1_flag } }
```

(*End of definition for* `\flag_new:n` *and others.*)

`\flag_show:n`
`\flag_log:n`
`\__flag_show:Nn`

To avoid changing the output here we mostly keep the old code.

```
\cs_new_protected:Npn \flag_show:n { \__flag_show:Nn \tl_show:n }
\cs_new_protected:Npn \flag_log:n { \__flag_show:Nn \tl_log:n }
\cs_new_protected:Npn \__flag_show:Nn #1#2
  {
    \exp_args:Nc \__kernel_chk_defined:NT { l_#2_flag }
      {
        \exp_args:Ne #1
          { \tl_to_str:n { flag~#2~height } = \flag_height:n {#2} }
      }
  }
```

(*End of definition for* `\flag_show:n`, `\flag_log:n`, *and* `\__flag_show:Nn`.)

```
⟨/package⟩
```

Chapter 61

l3clist implementation

The following test files are used for this code: m3clist002.

```
⟨*package⟩
⟨@@=clist⟩
```

\c_empty_clist

An empty comma list is simply an empty token list.

```
\cs_new_eq:NN \c_empty_clist \c_empty_tl
```

(*End of definition for* `\c_empty_clist`. *This variable is documented on page 193.*)

\l__clist_internal_clist

Scratch space for various internal uses. This comma list variable cannot be declared as such because it comes before `\clist_new:N`

```
\tl_new:N \l__clist_internal_clist
```

(*End of definition for* `\l__clist_internal_clist`.)

\s__clist_mark
\s__clist_stop

Internal scan marks.

```
\scan_new:N \s__clist_mark
\scan_new:N \s__clist_stop
```

(*End of definition for* `\s__clist_mark` *and* `\s__clist_stop`.)

__clist_use_none_delimit_by_s_mark:w
__clist_use_none_delimit_by_s_stop:w
__clist_use_i_delimit_by_s_stop:nw

Functions to gobble up to a scan mark.

```
\cs_new:Npn \__clist_use_none_delimit_by_s_mark:w #1 \s__clist_mark { }
\cs_new:Npn \__clist_use_none_delimit_by_s_stop:w #1 \s__clist_stop { }
\cs_new:Npn \__clist_use_i_delimit_by_s_stop:nw #1 #2 \s__clist_stop {#1}
```

(*End of definition for* `\__clist_use_none_delimit_by_s_mark:w`, `\__clist_use_none_delimit_by_s_stop:w`, *and* `\__clist_use_i_delimit_by_s_stop:nw`.)

__clist_tmp:w

A temporary function for various purposes.

```
\cs_new_protected:Npn \__clist_tmp:w { }
```

(*End of definition for* `\__clist_tmp:w`.)

61.1 Removing spaces around items

__clist_trim_next:w

Called as `\exp:w \__clist_trim_next:w \prg_do_nothing:` ⟨*comma list*⟩ ... it expands to `{`⟨*trimmed item*⟩`}` where the ⟨*trimmed item*⟩ is the first non-empty result from removing spaces from both ends of comma-delimited items in the ⟨*comma list*⟩. The `\prg_do_nothing:` marker avoids losing braces. The test for blank items is a somewhat optimized `\tl_if_empty:oTF` construction; if blank, another item is sought, otherwise trim spaces.

```
\cs_new:Npn \__clist_trim_next:w #1 ,
  {
    \tl_if_empty:oTF { \use_none:nn #1 ? }
      { \__clist_trim_next:w \prg_do_nothing: }
      { \tl_trim_spaces_apply:oN {#1} \exp_end: }
  }
```

(*End of definition for* `\__clist_trim_next:w`*.*)

__clist_sanitize:n
__clist_sanitize:Nn

The auxiliary `\__clist_sanitize:Nn` receives a delimiter (`\c_empty_tl` the first time, afterwards a comma) and that item as arguments. Unless we are done with the loop it calls `\__clist_wrap_item:w` to unbrace the item (using a comma delimiter is safe since `#2` came from removing spaces from an argument delimited by a comma) and possibly re-brace it if needed.

```
\cs_new:Npn \__clist_sanitize:n #1
  {
    \exp_after:wN \__clist_sanitize:Nn \exp_after:wN \c_empty_tl
    \exp:w \__clist_trim_next:w \prg_do_nothing:
    #1 , \s__clist_stop \prg_break: , \prg_break_point:
  }
\cs_new:Npn \__clist_sanitize:Nn #1#2
  {
    \__clist_use_none_delimit_by_s_stop:w #2 \s__clist_stop
    #1 \__clist_wrap_item:w #2 ,
    \exp_after:wN \__clist_sanitize:Nn \exp_after:wN ,
    \exp:w \__clist_trim_next:w \prg_do_nothing:
  }
```

(*End of definition for* `\__clist_sanitize:n` *and* `\__clist_sanitize:Nn`*.*)

__clist_if_wrap:n*TF*
__clist_if_wrap:w

True if the argument must be wrapped to avoid getting altered by some clist operations. That is the case whenever the argument

- starts or end with a space or contains a comma,
- is empty, or
- consists of a single braced group.

If the argument starts or ends with a space or contains a comma then one of the three arguments of `\__clist_if_wrap:w` will have its end delimiter (partly) in one of the three copies of `#1` in `\__clist_if_wrap:nTF`; this has a knock-on effect meaning that the result of the expansion is not empty; in that case, wrap. Otherwise, the argument is safe unless it starts with a brace group (or is empty) and it is empty or consists of a single `n`-type argument.

```
\prg_new_conditional:Npnn \__clist_if_wrap:n #1 { TF }
```

```
  {
    \tl_if_empty:oTF
      {
        \__clist_if_wrap:w
          \s__clist_mark ? #1 ~ \s__clist_mark ? ~ #1
          \s__clist_mark , ~ \s__clist_mark #1 ,
      }
      {
        \tl_if_head_is_group:nTF { #1 { } }
          {
            \tl_if_empty:nTF {#1}
              { \prg_return_true: }
              {
                \tl_if_empty:oTF { \use_none:n #1}
                  { \prg_return_true: }
                  { \prg_return_false: }
              }
          }
          { \prg_return_false: }
      }
      { \prg_return_true: }
  }
\cs_new:Npn \__clist_if_wrap:w #1 \s__clist_mark ? ~ #2 ~ \s__clist_mark #3 , { }
```

(*End of definition for* `\__clist_if_wrap:nTF` *and* `\__clist_if_wrap:w`.)

`\__clist_wrap_item:w`

Safe items are put in `\exp_not:n`, otherwise we put an extra set of braces.

```
\cs_new:Npn \__clist_wrap_item:w #1 ,
  { \__clist_if_wrap:nTF {#1} { \exp_not:n { {#1} } } { \exp_not:n {#1} } }
```

(*End of definition for* `\__clist_wrap_item:w`.)

61.2 Allocation and initialisation

`\clist_new:N`
`\clist_new:c`

Internally, comma lists are just token lists.

```
\cs_new_eq:NN \clist_new:N \tl_new:N
\cs_new_eq:NN \clist_new:c \tl_new:c
```

(*End of definition for* `\clist_new:N`. *This function is documented on page 184.*)

`\clist_const:Nn`
`\clist_const:Ne`
`\clist_const:Nx`
`\clist_const:cn`
`\clist_const:ce`
`\clist_const:cx`

Creating and initializing a constant comma list is done by sanitizing all items (stripping spaces and braces).

```
\cs_new_protected:Npn \clist_const:Nn #1#2
  { \tl_const:Ne #1 { \__clist_sanitize:n {#2} } }
\cs_generate_variant:Nn \clist_const:Nn { Ne , c , ce }
\cs_generate_variant:Nn \clist_const:Nn { Nx , cx }
```

(*End of definition for* `\clist_const:Nn`. *This function is documented on page 184.*)

`\clist_clear:N`
`\clist_clear:c`
`\clist_gclear:N`
`\clist_gclear:c`

Clearing comma lists is just the same as clearing token lists.

```
\cs_new_eq:NN \clist_clear:N  \tl_clear:N
\cs_new_eq:NN \clist_clear:c  \tl_clear:c
\cs_new_eq:NN \clist_gclear:N \tl_gclear:N
\cs_new_eq:NN \clist_gclear:c \tl_gclear:c
```

(*End of definition for* `\clist_clear:N` *and* `\clist_gclear:N`. *These functions are documented on page 184.*)

`\clist_clear_new:N`
`\clist_clear_new:c`
`\clist_gclear_new:N`
`\clist_gclear_new:c`

Once again a copy from the token list functions.

```
\cs_new_eq:NN \clist_clear_new:N  \tl_clear_new:N
\cs_new_eq:NN \clist_clear_new:c  \tl_clear_new:c
\cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
\cs_new_eq:NN \clist_gclear_new:c \tl_gclear_new:c
```

(*End of definition for* `\clist_clear_new:N` *and* `\clist_gclear_new:N`. *These functions are documented on page 184.*)

`\clist_set_eq:NN`
`\clist_set_eq:cN`
`\clist_set_eq:Nc`
`\clist_set_eq:cc`
`\clist_gset_eq:NN`
`\clist_gset_eq:cN`
`\clist_gset_eq:Nc`
`\clist_gset_eq:cc`

Once again, these are simple copies from the token list functions.

```
\cs_new_eq:NN \clist_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \clist_set_eq:Nc  \tl_set_eq:Nc
\cs_new_eq:NN \clist_set_eq:cN  \tl_set_eq:cN
\cs_new_eq:NN \clist_set_eq:cc  \tl_set_eq:cc
\cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
\cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
\cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
\cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc
```

(*End of definition for* `\clist_set_eq:NN` *and* `\clist_gset_eq:NN`. *These functions are documented on page 184.*)

`\clist_set_from_seq:NN`
`\clist_set_from_seq:cN`
`\clist_set_from_seq:Nc`
`\clist_set_from_seq:cc`
`\clist_gset_from_seq:NN`
`\clist_gset_from_seq:cN`
`\clist_gset_from_seq:Nc`
`\clist_gset_from_seq:cc`
`\__clist_set_from_seq:NNNN`
`\__clist_set_from_seq:n`

Setting a comma list from a comma-separated list is done using a simple mapping. Safe items are put in `\exp_not:n`, otherwise we put an extra set of braces. The first comma must be removed, except in the case of an empty comma-list.

```
\cs_new_protected:Npn \clist_set_from_seq:NN
  { \__clist_set_from_seq:NNNN \clist_clear:N  \__kernel_tl_set:Nx  }
\cs_new_protected:Npn \clist_gset_from_seq:NN
  { \__clist_set_from_seq:NNNN \clist_gclear:N \__kernel_tl_gset:Nx }
\cs_new_protected:Npn \__clist_set_from_seq:NNNN #1#2#3#4
  {
    \seq_if_empty:NTF #4
      { #1 #3 }
      {
        #2 #3
          {
            \exp_after:wN \use_none:n \exp:w \exp_end_continue_f:w
            \seq_map_function:NN #4 \__clist_set_from_seq:n
          }
      }
  }
\cs_new:Npn \__clist_set_from_seq:n #1
  {
    ,
    \__clist_if_wrap:nTF {#1}
      { \exp_not:n { {#1} } }
      { \exp_not:n {#1} }
  }
\cs_generate_variant:Nn \clist_set_from_seq:NN  {     Nc }
\cs_generate_variant:Nn \clist_set_from_seq:NN  { c , cc }
\cs_generate_variant:Nn \clist_gset_from_seq:NN {     Nc }
\cs_generate_variant:Nn \clist_gset_from_seq:NN { c , cc }
```

(*End of definition for* `\clist_set_from_seq:NN` *and others. These functions are documented on page 184.*)

`\clist_concat:NNN`
`\clist_concat:ccc`
`\clist_gconcat:NNN`
`\clist_gconcat:ccc`
`\__clist_concat:NNNN`

Concatenating comma lists is not quite as easy as it seems, as there needs to be the correct addition of a comma to the output. So a little work to do.

```
\cs_new_protected:Npn \clist_concat:NNN
  { \__clist_concat:NNNN \__kernel_tl_set:Nx }
\cs_new_protected:Npn \clist_gconcat:NNN
  { \__clist_concat:NNNN \__kernel_tl_gset:Nx }
\cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4
  {
    #1 #2
      {
        \exp_not:o #3
        \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
        \exp_not:o #4
      }
  }
\cs_generate_variant:Nn \clist_concat:NNN  { ccc }
\cs_generate_variant:Nn \clist_gconcat:NNN { ccc }
```

(*End of definition for* `\clist_concat:NNN`*,* `\clist_gconcat:NNN`*, and* `\__clist_concat:NNNN`*. These functions are documented on page 185.*)

`\clist_if_exist_p:N`
`\clist_if_exist_p:c`
`\clist_if_exist:NTF`
`\clist_if_exist:cTF`

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \clist_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \clist_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End of definition for* `\clist_if_exist:NTF`*. This function is documented on page 185.*)

61.3 Adding data to comma lists

`\clist_set:Nn`
`\clist_set:NV`
`\clist_set:Ne`
`\clist_set:No`
`\clist_set:Nx`
`\clist_set:cn`
`\clist_set:cV`
`\clist_set:ce`
`\clist_set:co`
`\clist_set:cx`
`\clist_gset:Nn`
`\clist_gset:NV`
`\clist_gset:Ne`
`\clist_gset:No`
`\clist_gset:Nx`
`\clist_gset:cn`
`\clist_gset:cV`
`\clist_gset:ce`
`\clist_gset:co`
`\clist_gset:cx`

```
\cs_new_protected:Npn \clist_set:Nn #1#2
  { \__kernel_tl_set:Nx #1 { \__clist_sanitize:n {#2} } }
\cs_new_protected:Npn \clist_gset:Nn #1#2
  { \__kernel_tl_gset:Nx #1 { \__clist_sanitize:n {#2} } }
\cs_generate_variant:Nn \clist_set:Nn  { NV , Ne , c , cV , ce }
\cs_generate_variant:Nn \clist_set:Nn  { No , Nx , co , cx }
\cs_generate_variant:Nn \clist_gset:Nn { NV , Ne , c , cV , ce }
\cs_generate_variant:Nn \clist_gset:Nn { No , Nx , co , cx }
```

(*End of definition for* `\clist_set:Nn` *and* `\clist_gset:Nn`*. These functions are documented on page 185.*)

`\clist_put_left:Nn`
`\clist_put_left:NV`
`\clist_put_left:Nv`
`\clist_put_left:Ne`
`\clist_put_left:No`
`\clist_put_left:Nx`
`\clist_put_left:cn`
`\clist_put_left:cV`
`\clist_put_left:cv`
`\clist_put_left:ce`
`\clist_put_left:co`
`\clist_put_left:cx`
`\clist_gput_left:Nn`
`\clist_gput_left:NV`
`\clist_gput_left:Nv`
`\clist_gput_left:Ne`
`\clist_gput_left:No`

Everything is based on concatenation after storing in `\l__clist_internal_clist`. This avoids having to worry here about space-trimming and so on.

```
\cs_new_protected:Npn \clist_put_left:Nn
  { \__clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }
\cs_new_protected:Npn \clist_gput_left:Nn
  { \__clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
\cs_new_protected:Npn \__clist_put_left:NNNn #1#2#3#4
```

```
  {
    #2 \l__clist_internal_clist {#4}
    #1 #3 \l__clist_internal_clist #3
  }
\cs_generate_variant:Nn \clist_put_left:Nn  { NV , Nv , Ne , c , cV , cv , ce }
\cs_generate_variant:Nn \clist_put_left:Nn  { No , Nx , co , cx }
\cs_generate_variant:Nn \clist_gput_left:Nn { NV , Nv , Ne , c , cV , cv , ce }
\cs_generate_variant:Nn \clist_gput_left:Nn { No , Nx , co , cx }
```

(*End of definition for* `\clist_put_left:Nn`, `\clist_gput_left:Nn`, *and* `\__clist_put_left:NNNn`. *These functions are documented on page* *185*.)

\clist_put_right:Nn
\clist_put_right:NV
\clist_put_right:Nv
\clist_put_right:Ne
\clist_put_right:No
\clist_put_right:Nx
\clist_put_right:cn
\clist_put_right:cV
\clist_put_right:cv
\clist_put_right:ce
\clist_put_right:co
\clist_put_right:cx
\clist_gput_right:Nn
\clist_gput_right:NV
\clist_gput_right:Nv
\clist_gput_right:Ne
\clist_gput_right:No
\clist_gput_right:Nx
\clist_gput_right:cn
\clist_gput_right:cV
\clist_gput_right:cv
\clist_gput_right:ce
\clist_gput_right:cx
\clist_gput_right:co
__clist_put_right:NNNn

```
\cs_new_protected:Npn \clist_put_right:Nn
  { \__clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
\cs_new_protected:Npn \clist_gput_right:Nn
  { \__clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
\cs_new_protected:Npn \__clist_put_right:NNNn #1#2#3#4
  {
    #2 \l__clist_internal_clist {#4}
    #1 #3 #3 \l__clist_internal_clist
  }
\cs_generate_variant:Nn \clist_put_right:Nn
  { NV , Nv , Ne , c , cV , cv , ce }
\cs_generate_variant:Nn \clist_put_right:Nn
  { No , Nx , co , cx }
\cs_generate_variant:Nn \clist_gput_right:Nn
  { NV , Nv , Ne , c , cV , cv , ce }
\cs_generate_variant:Nn \clist_gput_right:Nn
  { No , Nx , co , cx }
```

(*End of definition for* `\clist_put_right:Nn`, `\clist_gput_right:Nn`, *and* `\__clist_put_right:NNNn`. *These functions are documented on page* *185*.)

61.4 Comma lists as stacks

\clist_get:NN
\clist_get:cN
__clist_get:wN

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma. No need to trim spaces as comma-list *variables* are assumed to have "cleaned-up" items. (Note that grabbing a comma-delimited item removes an outer pair of braces if present, exactly as needed to uncover the underlying item.)

```
\cs_new_protected:Npn \clist_get:NN #1#2
  {
    \if_meaning:w #1 \c_empty_clist
      \tl_set:Nn #2 { \q_no_value }
    \else:
      \exp_after:wN \__clist_get:wN #1 , \s__clist_stop #2
    \fi:
  }
\cs_new_protected:Npn \__clist_get:wN #1 , #2 \s__clist_stop #3
  { \tl_set:Nn #3 {#1} }
\cs_generate_variant:Nn \clist_get:NN { c }
```

(*End of definition for* `\clist_get:NN` *and* `\__clist_get:wN`. *This function is documented on page* *191*.)

\clist_pop:NN
\clist_pop:cN
\clist_gpop:NN
\clist_gpop:cN
__clist_pop:NNN
__clist_pop:wwNNN
__clist_pop:wN

An empty clist leads to `\q_no_value`, otherwise grab until the first comma and assign to the variable. The second argument of `\__clist_pop:wwNNN` is a comma list ending in a comma and `\s__clist_mark`, unless the original clist contained exactly one item: then the argument is just `\s__clist_mark`. The next auxiliary picks either `\exp_not:n` or `\use_none:n` as #2, ensuring that the result can safely be an empty comma list.

```
\cs_new_protected:Npn \clist_pop:NN
  { \__clist_pop:NNN \__kernel_tl_set:Nx }
\cs_new_protected:Npn \clist_gpop:NN
  { \__clist_pop:NNN \__kernel_tl_gset:Nx }
\cs_new_protected:Npn \__clist_pop:NNN #1#2#3
  {
    \if_meaning:w #2 \c_empty_clist
      \tl_set:Nn #3 { \q_no_value }
    \else:
      \exp_after:wN \__clist_pop:wwNNN #2 , \s__clist_mark \s__clist_stop #1#2#3
    \fi:
  }
\cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \s__clist_stop #3#4#5
  {
    \tl_set:Nn #5 {#1}
    #3 #4
      {
        \__clist_pop:wN \prg_do_nothing:
          #2 \exp_not:o
          , \s__clist_mark \use_none:n
        \s__clist_stop
      }
  }
\cs_new:Npn \__clist_pop:wN #1 , \s__clist_mark #2 #3 \s__clist_stop { #2 {#1} }
\cs_generate_variant:Nn \clist_pop:NN  { c }
\cs_generate_variant:Nn \clist_gpop:NN { c }
```

(*End of definition for* `\clist_pop:NN` *and others. These functions are documented on page* *191*.)

\clist_get:NN*TF*
\clist_get:cN*TF*
\clist_pop:NN*TF*
\clist_pop:cN*TF*
\clist_gpop:NN*TF*
\clist_gpop:cN*TF*
__clist_pop_TF:NNN

The same, as branching code: very similar to the above.

```
\prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
  {
    \if_meaning:w #1 \c_empty_clist
      \prg_return_false:
    \else:
      \exp_after:wN \__clist_get:wN #1 , \s__clist_stop #2
      \prg_return_true:
    \fi:
  }
\prg_generate_conditional_variant:Nnn \clist_get:NN { c } { T , F , TF }
\prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
  { \__clist_pop_TF:NNN \__kernel_tl_set:Nx #1 #2 }
\prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
  { \__clist_pop_TF:NNN \__kernel_tl_gset:Nx #1 #2 }
\cs_new_protected:Npn \__clist_pop_TF:NNN #1#2#3
  {
    \if_meaning:w #2 \c_empty_clist
      \prg_return_false:
    \else:
```

```
      \exp_after:wN \__clist_pop:wwNNN #2 , \s__clist_mark \s__clist_stop #1#2#3
      \prg_return_true:
    \fi:
  }
\prg_generate_conditional_variant:Nnn \clist_pop:NN { c } { T , F , TF }
\prg_generate_conditional_variant:Nnn \clist_gpop:NN { c } { T , F , TF }
```

(*End of definition for* `\clist_get:NNTF` *and others. These functions are documented on page 191.*)

`\clist_push:Nn`
`\clist_push:NV`
`\clist_push:No`
`\clist_push:Nx`
`\clist_push:cn`
`\clist_push:cV`
`\clist_push:co`
`\clist_push:cx`
`\clist_gpush:Nn`
`\clist_gpush:NV`
`\clist_gpush:No`
`\clist_gpush:Nx`
`\clist_gpush:cn`
`\clist_gpush:cV`
`\clist_gpush:co`
`\clist_gpush:cx`

Pushing to a comma list is the same as adding on the left.

```
\cs_new_eq:NN \clist_push:Nn  \clist_put_left:Nn
\cs_generate_variant:Nn \clist_push:Nn { NV , No , Nx , c , cV , co , cx }
\cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
\cs_generate_variant:Nn \clist_gpush:Nn { NV , No , Nx , c , cV , co , cx }
```

(*End of definition for* `\clist_push:Nn` *and* `\clist_gpush:Nn`*. These functions are documented on page 192.*)

61.5 Modifying comma lists

`\l__clist_internal_remove_clist`
`\l__clist_internal_remove_seq`

An internal comma list and a sequence for the removal routines.

```
\clist_new:N \l__clist_internal_remove_clist
\seq_new:N \l__clist_internal_remove_seq
```

(*End of definition for* `\l__clist_internal_remove_clist` *and* `\l__clist_internal_remove_seq`*.*)

`\clist_remove_duplicates:N`
`\clist_remove_duplicates:c`
`\clist_gremove_duplicates:N`
`\clist_gremove_duplicates:c`
`\__clist_remove_duplicates:NN`

Removing duplicates means making a new list then copying it.

```
\cs_new_protected:Npn \clist_remove_duplicates:N
  { \__clist_remove_duplicates:NN \clist_set_eq:NN }
\cs_new_protected:Npn \clist_gremove_duplicates:N
  { \__clist_remove_duplicates:NN \clist_gset_eq:NN }
\cs_new_protected:Npn \__clist_remove_duplicates:NN #1#2
  {
    \clist_clear:N \l__clist_internal_remove_clist
    \clist_map_inline:Nn #2
      {
        \clist_if_in:NnF \l__clist_internal_remove_clist {##1}
          {
            \tl_put_right:Ne \l__clist_internal_remove_clist
              {
                \clist_if_empty:NF \l__clist_internal_remove_clist { , }
                \__clist_if_wrap:nTF {##1} { \exp_not:n { {##1} } } { \exp_not:n {##1} }
              }
          }
      }
    #1 #2 \l__clist_internal_remove_clist
  }
\cs_generate_variant:Nn \clist_remove_duplicates:N  { c }
\cs_generate_variant:Nn \clist_gremove_duplicates:N { c }
```

(*End of definition for* `\clist_remove_duplicates:N`*,* `\clist_gremove_duplicates:N`*, and* `\__clist_remove_duplicates:NN`*. These functions are documented on page 186.*)

\clist_remove_all:Nn
\clist_remove_all:cn
\clist_remove_all:NV
\clist_remove_all:cV
\clist_remove_all:Ne
\clist_remove_all:ce
\clist_gremove_all:Nn
\clist_gremove_all:cn
\clist_gremove_all:NV
\clist_gremove_all:cV
\clist_gremove_all:Ne
\clist_gremove_all:ce
__clist_remove_all:NNNn
__clist_remove_all:w
__clist_remove_all:

The method used here for safe items is very similar to `\tl_replace_all:Nnn`. However, if the item contains commas or leading/trailing spaces, or is empty, or consists of a single brace group, we know that it can only appear within braces so the code would fail; instead just convert to a sequence and do the removal with l3seq code (it involves somewhat elaborate code to do most of the work expandably but the final token list comparisons non-expandably).

For "safe" items, build a function delimited by the ⟨*item*⟩ that should be removed, surrounded with commas, and call that function followed by the expanded comma list, and another copy of the ⟨*item*⟩. The loop is controlled by the argument grabbed by `\__clist_remove_all:w`: when the item was found, the `\s__clist_mark` delimiter used is the one inserted by `\__clist_tmp:w`, and `\__clist_use_none_delimit_by_s_stop:w` is deleted. At the end, the final ⟨*item*⟩ is grabbed, and the argument of `\__clist_tmp:w` contains `\s__clist_mark`: in that case, `\__clist_remove_all:w` removes the second `\s__clist_mark` (inserted by `\__clist_tmp:w`), and lets `\__clist_use_none_delimit_by_s_stop:w` act.

No brace is lost because items are always grabbed with a leading comma. The result of the first assignment has an extra leading comma, which we remove in a second assignment. Two exceptions: if the clist lost all of its elements, the result is empty, and we shouldn't remove anything; if the clist started up empty, the first step happens to turn it into a single comma, and the second step removes it.

```
\cs_new_protected:Npn \clist_remove_all:Nn
  { \__clist_remove_all:NNNn \clist_set_from_seq:NN \__kernel_tl_set:Nx }
\cs_new_protected:Npn \clist_gremove_all:Nn
  { \__clist_remove_all:NNNn \clist_gset_from_seq:NN \__kernel_tl_gset:Nx }
\cs_new_protected:Npn \__clist_remove_all:NNNn #1#2#3#4
  {
    \__clist_if_wrap:nTF {#4}
      {
        \seq_set_from_clist:NN \l__clist_internal_remove_seq #3
        \seq_remove_all:Nn \l__clist_internal_remove_seq {#4}
        #1 #3 \l__clist_internal_remove_seq
      }
      {
        \cs_set:Npn \__clist_tmp:w ##1 , #4 ,
          {
            ##1
            , \s__clist_mark , \__clist_use_none_delimit_by_s_stop:w ,
            \__clist_remove_all:
          }
        #2 #3
          {
            \exp_after:wN \__clist_remove_all:
            #3 , \s__clist_mark , #4 , \s__clist_stop
          }
        \clist_if_empty:NF #3
          {
            #2 #3
              {
                \exp_args:No \exp_not:o
                  { \exp_after:wN \use_none:n #3 }
              }
          }
```

```
      }
  }
\cs_new:Npn \__clist_remove_all:
  { \exp_after:wN \__clist_remove_all:w \__clist_tmp:w , }
\cs_new:Npn \__clist_remove_all:w #1 , \s__clist_mark , #2 , { \exp_not:n {#1} }
\cs_generate_variant:Nn \clist_remove_all:Nn  { c , NV , cV , Ne , ce }
\cs_generate_variant:Nn \clist_gremove_all:Nn { c , NV , cV , Ne , ce }
```

(*End of definition for* `\clist_remove_all:Nn` *and others. These functions are documented on page 186.*)

`\clist_reverse:N`
`\clist_reverse:c`
`\clist_greverse:N`
`\clist_greverse:c`

Use `\clist_reverse:n` in an `e`-expanding assignment. The extra work that `\clist_reverse:n` does to preserve braces and spaces would not be needed for the well-controlled case of `N`-type comma lists, but the slow-down is not too bad.

```
\cs_new_protected:Npn \clist_reverse:N #1
  { \__kernel_tl_set:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
\cs_new_protected:Npn \clist_greverse:N #1
  { \__kernel_tl_gset:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
\cs_generate_variant:Nn \clist_reverse:N { c }
\cs_generate_variant:Nn \clist_greverse:N { c }
```

(*End of definition for* `\clist_reverse:N` *and* `\clist_greverse:N`*. These functions are documented on page 186.*)

`\clist_reverse:n`
`\__clist_reverse:wwNww`
`\__clist_reverse_end:ww`

The reversed token list is built one item at a time, and stored between `\s__clist_stop` and `\s__clist_mark`, in the form of `?` followed by zero or more instances of "⟨*item*⟩`,`". We start from a comma list "$\langle item_1\rangle$`,`...`,`$\langle item_n\rangle$". During the loop, the auxiliary `\__clist_reverse:wwNww` receives "`?`$\langle item_i\rangle$" as `#1`, "$\langle item_{i+1}\rangle$`,`...`,`$\langle item_n\rangle$" as `#2`, `\__clist_reverse:wwNww` as `#3`, what remains until `\s__clist_stop` as `#4`, and "$\langle item_{i-1}\rangle$`,`...`,`$\langle item_1\rangle$`,`" as `#5`. The auxiliary moves `#1` just before `#5`, with a comma, and calls itself (`#3`). After the last item is moved, `\__clist_reverse:wwNww` receives "`\s__clist_mark \__clist_reverse:wwNww !`" as its argument `#1`, thus `\__clist_reverse_end:ww` as its argument `#3`. This second auxiliary cleans up until the marker `!`, removes the trailing comma (introduced when the first item was moved after `\s__clist_stop`), and leaves its argument `#1` within `\exp_not:n`. There is also a need to remove a leading comma, hence `\exp_not:o` and `\use_none:n`.

```
\cs_new:Npn \clist_reverse:n #1
  {
    \__clist_reverse:wwNww ? #1 ,
      \s__clist_mark \__clist_reverse:wwNww ! ,
      \s__clist_mark \__clist_reverse_end:ww
      \s__clist_stop ? \s__clist_mark
  }
\cs_new:Npn \__clist_reverse:wwNww
    #1 , #2 \s__clist_mark #3 #4 \s__clist_stop ? #5 \s__clist_mark
  { #3 ? #2 \s__clist_mark #3 #4 \s__clist_stop #1 , #5 \s__clist_mark }
\cs_new:Npn \__clist_reverse_end:ww #1 ! #2 , \s__clist_mark
  { \exp_not:o { \use_none:n #2 } }
```

(*End of definition for* `\clist_reverse:n`*,* `\__clist_reverse:wwNww`*, and* `\__clist_reverse_end:ww`*. This function is documented on page 186.*)

`\clist_sort:Nn`
`\clist_sort:cn`
`\clist_gsort:Nn`
`\clist_gsort:cn`

Implemented in l3sort.

(*End of definition for* `\clist_sort:Nn` *and* `\clist_gsort:Nn`*. These functions are documented on page 187.*)

61.6 Comma list conditionals

\clist_if_empty_p:N
\clist_if_empty_p:c
\clist_if_empty:NTF
\clist_if_empty:cTF

Simple copies from the token list variable material.

```
\prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N
  { p , T , F , TF }
\prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c
  { p , T , F , TF }
```

(*End of definition for* `\clist_if_empty:NTF`. *This function is documented on page 187.*)

\clist_if_empty_p:n
\clist_if_empty:nTF
__clist_if_empty_n:w
__clist_if_empty_n:wNw

As usual, we insert a token (here `?`) before grabbing any argument: this avoids losing braces. The argument of `\tl_if_empty:oTF` is empty if `#1` is `?` followed by blank spaces (besides, this particular variant of the emptiness test is optimized). If the item of the comma list is blank, grab the next one. As soon as one item is non-blank, exit: the second auxiliary grabs `\prg_return_false:` as `#2`, unless every item in the comma list was blank and the loop actually got broken by the trailing `\s__clist_mark \prg_return_false:` item.

```
\prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
  {
    \__clist_if_empty_n:w ? #1
    , \s__clist_mark \prg_return_false:
    , \s__clist_mark \prg_return_true:
    \s__clist_stop
  }
\cs_new:Npn \__clist_if_empty_n:w #1 ,
  {
    \tl_if_empty:oTF { \use_none:nn #1 ? }
      { \__clist_if_empty_n:w ? }
      { \__clist_if_empty_n:wNw }
  }
\cs_new:Npn \__clist_if_empty_n:wNw #1 \s__clist_mark #2#3 \s__clist_stop {#2}
```

(*End of definition for* `\clist_if_empty:nTF`*,* `\__clist_if_empty_n:w`*, and* `\__clist_if_empty_n:wNw`*. This function is documented on page 187.*)

\clist_if_in:NnTF
\clist_if_in:NVTF
\clist_if_in:NoTF
\clist_if_in:cnTF
\clist_if_in:cVTF
\clist_if_in:coTF
\clist_if_in:nnTF
\clist_if_in:nVTF
\clist_if_in:noTF
__clist_if_in_return:nnN

For "safe" items, we simply surround the comma list, and the item, with commas, then use the same code as for `\tl_if_in:Nn`. For "unsafe" items we follow the same route as `\seq_if_in:Nn`, mapping through the list a comparison function. If found, return `true` and remove `\prg_return_false:`.

```
\prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T   , F , TF }
  {
    \exp_args:No \__clist_if_in_return:nnN #1 {#2} #1
  }
\prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T   , F , TF }
  {
    \clist_set:Nn \l__clist_internal_clist {#1}
    \exp_args:No \__clist_if_in_return:nnN \l__clist_internal_clist {#2}
      \l__clist_internal_clist
  }
\cs_new_protected:Npn \__clist_if_in_return:nnN #1#2#3
  {
    \__clist_if_wrap:nTF {#2}
      {
        \cs_set:Npe \__clist_tmp:w ##1
```

```
            {
              \exp_not:N \tl_if_eq:nnT {##1}
              \exp_not:n
                {
                  {#2}
                  { \clist_map_break:n { \prg_return_true: \use_none:n } }
                }
            }
          \clist_map_function:NN #3 \__clist_tmp:w
          \prg_return_false:
        }
        {
          \cs_set:Npn \__clist_tmp:w ##1 ,#2, { }
          \tl_if_empty:oTF
            { \__clist_tmp:w ,#1, {} {} ,#2, }
            { \prg_return_false: } { \prg_return_true: }
        }
    }
\prg_generate_conditional_variant:Nnn \clist_if_in:Nn
  { NV , No , c , cV , co } { T , F , TF }
\prg_generate_conditional_variant:Nnn \clist_if_in:nn
  { nV , no } { T , F , TF }
```

(*End of definition for* `\clist_if_in:NnTF` *,* `\clist_if_in:nnTF` *, and* `\__clist_if_in_return:nnN`*. These functions are documented on page* *187**.*)

61.7 Mapping over comma lists

`\clist_map_function:NN`
`\clist_map_function:cN`
`\__clist_map_function:Nw`
`\__clist_map_function_end:w`

If the variable is empty, the mapping is skipped (otherwise, that comma-list would be seen as consisting of one empty item). Then loop over the comma-list, grabbing eight comma-delimited items at a time. The end is marked by `\s__clist_stop`, which may not appear in any of the items. Once the last group of eight items has been reached, we go through them more slowly using `\__clist_map_function_end:w`. The auxiliary function `\__clist_map_function:Nw` is also used in some other clist mappings.

```
\cs_new:Npn \clist_map_function:NN #1#2
  {
    \clist_if_empty:NF #1
      {
        \exp_after:wN \__clist_map_function:Nw \exp_after:wN #2 #1 ,
          \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
          \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
        \prg_break_point:Nn \clist_map_break: { }
      }
  }
\cs_new:Npn \__clist_map_function:Nw #1 #2, #3, #4, #5, #6, #7, #8, #9,
  {
    \__clist_use_none_delimit_by_s_stop:w
      #9 \__clist_map_function_end:w \s__clist_stop
    #1 {#2} #1 {#3} #1 {#4} #1 {#5} #1 {#6} #1 {#7} #1 {#8} #1 {#9}
    \__clist_map_function:Nw #1
  }
\cs_new:Npn \__clist_map_function_end:w \s__clist_stop #1#2
  {
```

```
    \__clist_use_none_delimit_by_s_stop:w #2 \clist_map_break: \s__clist_stop
    #1 {#2}
    \__clist_map_function_end:w \s__clist_stop
  }
\cs_generate_variant:Nn \clist_map_function:NN { c }
```

(*End of definition for* `\clist_map_function:NN`*,* `\__clist_map_function:Nw`*, and* `\__clist_map_function_end:w`*. This function is documented on page 188.*)

`\clist_map_function:nN`
`\clist_map_function:eN`
`\__clist_map_function_n:Nn`
`\__clist_map_unbrace:wn`

The `n`-type mapping function is a bit more awkward, since spaces must be trimmed from each item. Space trimming is again based on `\__clist_trim_next:w`. The auxiliary `\__clist_map_function_n:Nn` receives as arguments the function, and the next non-empty item (after space trimming but before brace removal). One level of braces is removed by `\__clist_map_unbrace:wn`.

```
\cs_new:Npn \clist_map_function:nN #1#2
  {
    \exp_after:wN \__clist_map_function_n:Nn \exp_after:wN #2
    \exp:w \__clist_trim_next:w \prg_do_nothing: #1 ,
      \s__clist_stop \clist_map_break: ,
    \prg_break_point:Nn \clist_map_break: { }
  }
\cs_generate_variant:Nn \clist_map_function:nN { e }
\cs_new:Npn \__clist_map_function_n:Nn #1 #2
  {
    \__clist_use_none_delimit_by_s_stop:w #2 \s__clist_stop
    \__clist_map_unbrace:wn #2 , #1
    \exp_after:wN \__clist_map_function_n:Nn \exp_after:wN #1
    \exp:w \__clist_trim_next:w \prg_do_nothing:
  }
\cs_new:Npn \__clist_map_unbrace:wn #1, #2 { #2 {#1} }
```

(*End of definition for* `\clist_map_function:nN`*,* `\__clist_map_function_n:Nn`*, and* `\__clist_map_unbrace:wn`*. This function is documented on page 188.*)

`\clist_map_inline:Nn`
`\clist_map_inline:cn`
`\clist_map_inline:nn`

Inline mapping is done by creating a suitable function "on the fly": this is done globally to avoid any issues with TeX's groups. We use a different function for each level of nesting.

Since the mapping is non-expandable, we can perform the space-trimming needed by the `n` version simply by storing the comma-list in a variable. We don't need a different comma-list for each nesting level: the comma-list is expanded before the mapping starts.

```
\cs_new_protected:Npn \clist_map_inline:Nn #1#2
  {
    \clist_if_empty:NF #1
      {
        \int_gincr:N \g__kernel_prg_map_int
        \cs_gset_protected:cpn
          { __clist_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
        \exp_last_unbraced:Nco \__clist_map_function:Nw
          { __clist_map_ \int_use:N \g__kernel_prg_map_int :w }
          #1 ,
          \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
          \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
        \prg_break_point:Nn \clist_map_break:
          { \int_gdecr:N \g__kernel_prg_map_int }
```

```
      }
  }
\cs_new_protected:Npn \clist_map_inline:nn #1
  {
    \clist_set:Nn \l__clist_internal_clist {#1}
    \clist_map_inline:Nn \l__clist_internal_clist
  }
\cs_generate_variant:Nn \clist_map_inline:Nn { c }
```

(*End of definition for* `\clist_map_inline:Nn` *and* `\clist_map_inline:nn`*. These functions are documented on page 188.*)

`\clist_map_variable:NNn`
`\clist_map_variable:cNn`
`\clist_map_variable:nNn`
`\__clist_map_variable:Nnn`

The `N`-type version is a straightforward application of `\clist_map_tokens:Nn`, calling `\__clist_map_variable:Nnn` for each item to assign the variable and run the user's code. The `n`-type version is *not* implemented in terms of the `n`-type function `\clist_map_tokens:Nn`, because here we are allowed to clean up the `n`-type comma list non-expandably.

```
\cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
  { \clist_map_tokens:Nn #1 { \__clist_map_variable:Nnn #2 {#3} } }
\cs_generate_variant:Nn \clist_map_variable:NNn { c }
\cs_new_protected:Npn \__clist_map_variable:Nnn #1#2#3
  { \tl_set:Nn #1 {#3} #2 }
\cs_new_protected:Npn \clist_map_variable:nNn #1
  {
    \clist_set:Nn \l__clist_internal_clist {#1}
    \clist_map_variable:NNn \l__clist_internal_clist
  }
```

(*End of definition for* `\clist_map_variable:NNn`*,* `\clist_map_variable:nNn`*, and* `\__clist_map_variable:Nnn`*. These functions are documented on page 188.*)

`\clist_map_tokens:Nn`
`\clist_map_tokens:cn`
`\__clist_map_tokens:nw`
`\__clist_map_tokens_end:w`

Essentially a copy of `\clist_map_function:NN` with braces added.

```
\cs_new:Npn \clist_map_tokens:Nn #1#2
  {
    \clist_if_empty:NF #1
      {
        \exp_last_unbraced:Nno \__clist_map_tokens:nw {#2} #1 ,
          \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
          \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
        \prg_break_point:Nn \clist_map_break: { }
      }
  }
\cs_new:Npn \__clist_map_tokens:nw #1 #2, #3, #4, #5, #6, #7, #8, #9,
  {
    \__clist_use_none_delimit_by_s_stop:w
      #9 \__clist_map_tokens_end:w \s__clist_stop
    \use:n {#1} {#2} \use:n {#1} {#3} \use:n {#1} {#4} \use:n {#1} {#5}
    \use:n {#1} {#6} \use:n {#1} {#7} \use:n {#1} {#8} \use:n {#1} {#9}
    \__clist_map_tokens:nw {#1}
  }
\cs_new:Npn \__clist_map_tokens_end:w \s__clist_stop \use:n #1#2
  {
    \__clist_use_none_delimit_by_s_stop:w #2 \clist_map_break: \s__clist_stop
    #1 {#2}
```

```
    \__clist_map_tokens_end:w \s__clist_stop
  }
\cs_generate_variant:Nn \clist_map_tokens:Nn { c }
```

(*End of definition for* `\clist_map_tokens:Nn`*,* `\__clist_map_tokens:nw`*, and* `\__clist_map_tokens_end:w`*. This function is documented on page 188.*)

`\clist_map_tokens:nn`
`\__clist_map_tokens_n:nw`

Similar to `\clist_map_function:nN` but with a different way of grabbing items because we cannot use `\exp_after:wN` to pass the ⟨*code*⟩.

```
\cs_new:Npn \clist_map_tokens:nn #1#2
  {
    \__clist_map_tokens_n:nw {#2}
    \prg_do_nothing: #1 , \s__clist_stop \clist_map_break: ,
    \prg_break_point:Nn \clist_map_break: { }
  }
\cs_new:Npn \__clist_map_tokens_n:nw #1#2 ,
  {
    \tl_if_empty:oF { \use_none:nn #2 ? }
      {
        \__clist_use_none_delimit_by_s_stop:w #2 \s__clist_stop
        \tl_trim_spaces_apply:oN {#2} \use_ii_i:nn
        \__clist_map_unbrace:wn , {#1}
      }
    \__clist_map_tokens_n:nw {#1} \prg_do_nothing:
  }
```

(*End of definition for* `\clist_map_tokens:nn` *and* `\__clist_map_tokens_n:nw`*. This function is documented on page 188.*)

`\clist_map_break:`
`\clist_map_break:n`

The break statements use the general `\prg_map_break:Nn` mechanism.

```
\cs_new:Npn \clist_map_break:
  { \prg_map_break:Nn \clist_map_break: { } }
\cs_new:Npn \clist_map_break:n
  { \prg_map_break:Nn \clist_map_break: }
```

(*End of definition for* `\clist_map_break:` *and* `\clist_map_break:n`*. These functions are documented on page 188.*)

`\clist_count:N`
`\clist_count:c`
`\clist_count:n`
`\clist_count:e`
`\__clist_count:n`
`\__clist_count:w`

Counting the items in a comma list is done using the same approach as for other token count functions: turn each entry into a `+1` then use integer evaluation to actually do the mathematics. In the case of an `n`-type comma-list, we could of course use `\clist_map_function:nN`, but that is very slow, because it carefully removes spaces. Instead, we loop manually, and skip blank items (but not `{}`, hence the extra spaces).

```
\cs_new:Npn \clist_count:N #1
  {
    \int_eval:n
      {
        0
        \clist_map_function:NN #1 \__clist_count:n
      }
  }
\cs_generate_variant:Nn \clist_count:N { c }
\cs_new:Npn \__clist_count:n #1 { + 1 }
\cs_set_protected:Npn \__clist_tmp:w #1
  {
```

```
    \cs_new:Npn \clist_count:n ##1
      {
        \int_eval:n
          {
            0
            \__clist_count:w #1
            ##1 , \s__clist_stop \prg_break: , \prg_break_point:
          }
      }
    \cs_new:Npn \__clist_count:w ##1 ,
      {
        \__clist_use_none_delimit_by_s_stop:w ##1 \s__clist_stop
        \tl_if_blank:nF {##1} { + 1 }
        \__clist_count:w #1
      }
  }
\exp_args:No \__clist_tmp:w \c_space_tl
\cs_generate_variant:Nn \clist_count:n { e }
```

(*End of definition for* `\clist_count:N` *and others. These functions are documented on page 189.*)

61.8 Using comma lists

`\clist_use:Nnnn`
`\clist_use:cnnn`
`\__clist_use:wwn`
`\__clist_use:nwwwwnwn`
`\__clist_use:nwwn`
`\clist_use:Nn`
`\clist_use:cn`

First check that the variable exists. Then count the items in the comma list. If it has none, output nothing. If it has one item, output that item, brace stripped (note that space-trimming has already been done when the comma list was assigned). If it has two, place the ⟨*separator between two*⟩ in the middle.

Otherwise, `\__clist_use:nwwwwnwn` takes the following arguments; 1: a ⟨*separator*⟩, 2, 3, 4: three items from the comma list (or quarks), 5: the rest of the comma list, 6: a ⟨*continuation*⟩ function (`use_ii` or `use_iii` with its ⟨*separator*⟩ argument), 7: junk, and 8: the temporary result, which is built in a brace group following `\s__clist_stop`. The ⟨*separator*⟩ and the first of the three items are placed in the result, then we use the ⟨*continuation*⟩, placing the remaining two items after it. When we begin this loop, the three items really belong to the comma list, the first `\s__clist_mark` is taken as a delimiter to the `use_ii` function, and the continuation is `use_ii` itself. When we reach the last two items of the original token list, `\s__clist_mark` is taken as a third item, and now the second `\s__clist_mark` serves as a delimiter to `use_ii`, switching to the other ⟨*continuation*⟩, `use_iii`, which uses the ⟨*separator between final two*⟩.

```
\cs_new:Npn \clist_use:Nnnn #1#2#3#4
  {
    \clist_if_exist:NTF #1
      {
        \int_case:nnF { \clist_count:N #1 }
          {
            { 0 } { }
            { 1 } { \exp_after:wN \__clist_use:wwn #1 , , { } }
            { 2 } { \exp_after:wN \__clist_use:wwn #1 , {#2} }
          }
          {
            \exp_after:wN \__clist_use:nwwwwnwn
            \exp_after:wN { \exp_after:wN } #1 ,
            \s__clist_mark , { \__clist_use:nwwwwnwn {#3} }
```

```
            \s__clist_mark , { \__clist_use:nwwn {#4} }
            \s__clist_stop { }
          }
      }
      {
        \msg_expandable_error:nnn
          { kernel } { bad-variable } {#1}
      }
  }
\cs_generate_variant:Nn \clist_use:Nnnn { c }
\cs_new:Npn \__clist_use:wwn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
\cs_new:Npn \__clist_use:nwwwwnwn
    #1#2 , #3 , #4 , #5 \s__clist_mark , #6#7 \s__clist_stop #8
  { #6 {#3} , {#4} , #5 \s__clist_mark , {#6} #7 \s__clist_stop { #8 #1 #2 } }
\cs_new:Npn \__clist_use:nwwn #1#2 , #3 \s__clist_stop #4
  { \exp_not:n { #4 #1 #2 } }
\cs_new:Npn \clist_use:Nn #1#2
  { \clist_use:Nnnn #1 {#2} {#2} {#2} }
\cs_generate_variant:Nn \clist_use:Nn { c }
```

(*End of definition for* `\clist_use:Nnnn` *and others. These functions are documented on page 190.*)

`\clist_use:nnnn`
`\clist_use:nn`
`\__clist_use:Nw`
`\__clist_use_one:w`
`\__clist_use_end:w`
`\__clist_use_more:w`

Items are grabbed by `\__clist_use:Nw`, which detects blank items with a `\tl_if_-empty:oTF` test (in which case it recurses). Non-blank items are either the end of the list, in which case the argument `#1` of `\__clist_use:Nw` is used to properly end the list, or are normal items, which must be trimmed and properly unbraced. As we find successive items, the long list of `\__clist_use:Nw` calls gets shortened and we end up calling `\__clist_use_more:w` once we have found 3 items. This auxiliary leaves the first-found item and the general separator, and calls `\__clist_use:Nw` to find more items. A subtlety is that we use `\__clist_use_end:w` both in the case of a two-item list and for the last two items of a general list: to get the correct separator, `\__clist_use_more:w` replaces the separator-of-two by the last-separator when called, namely as soon as we have found three items.

```
\cs_new:Npn \clist_use:nnnn #1#2#3#4
  {
    \__clist_use:Nw \__clist_use_none_delimit_by_s_stop:w
    \__clist_use:Nw \__clist_use_one:w
    \__clist_use:Nw \__clist_use_end:w
    \__clist_use_more:w ;
      {#2} {#3} {#4} ;
    \prg_do_nothing: #1 , \s__clist_mark ,
    \s__clist_stop
  }
\cs_new:Npn \__clist_use:Nw #1#2 ; #3 ; #4 ,
  {
    \tl_if_empty:oTF { \use_none:nn #4 ? }
      { \__clist_use:Nw #1#2 ; }
      {
        \__clist_use_none_delimit_by_s_mark:w #4 #1 \s__clist_mark
        \tl_trim_spaces_apply:oN {#4} \use_ii_i:nn
        \__clist_map_unbrace:wn , { #2 ; }
      }
    #3 ; \prg_do_nothing:
```

```
  }
\cs_new:Npn \__clist_use_one:w \s__clist_mark #1 , #2#3#4 \s__clist_stop
  { \exp_not:n {#3} }
\cs_new:Npn \__clist_use_end:w
    \s__clist_mark #1 , #2#3#4#5#6 \s__clist_stop
  { \exp_not:n { #4 #5 #3 } }
\cs_new:Npn \__clist_use_more:w ; #1#2#3#4#5#6 ;
  {
    \exp_not:n { #3 #5 }
    \__clist_use:Nw \__clist_use_end:w \__clist_use_more:w ;
    {#1} {#2} {#6} {#5} {#6} ;
  }
\cs_new:Npn \clist_use:nn #1#2 { \clist_use:nnnn {#1} {#2} {#2} {#2} }
```

(*End of definition for* `\clist_use:nnnn` *and others. These functions are documented on page 191.*)

61.9 Using a single item

`\clist_item:Nn`
`\clist_item:cn`
`\__clist_item:nnnN`
`\__clist_item:ffoN`
`\__clist_item:ffnN`
`\__clist_item_N_loop:nw`

To avoid needing to test the end of the list at each step, we first compute the ⟨*length*⟩ of the list. If the item number is 0, less than $-\langle length\rangle$, or more than ⟨*length*⟩, the result is empty. If it is negative, but not less than $-\langle length\rangle$, add $\langle length\rangle + 1$ to the item number before performing the loop. The loop itself is very simple, return the item if the counter reached 1, otherwise, decrease the counter and repeat.

```
\cs_new:Npn \clist_item:Nn #1#2
  {
    \__clist_item:ffoN
      { \clist_count:N #1 }
      { \int_eval:n {#2} }
      #1
      \__clist_item_N_loop:nw
  }
\cs_new:Npn \__clist_item:nnnN #1#2#3#4
  {
    \int_compare:nNnTF {#2} < 0
      {
        \int_compare:nNnTF {#2} < { - #1 }
          { \__clist_use_none_delimit_by_s_stop:w }
          { \exp_args:Nf #4 { \int_eval:n { #2 + 1 + #1 } } }
      }
      {
        \int_compare:nNnTF {#2} > {#1}
          { \__clist_use_none_delimit_by_s_stop:w }
          { #4 {#2} }
      }
    { } , #3 , \s__clist_stop
  }
\cs_generate_variant:Nn \__clist_item:nnnN { ffo, ff }
\cs_new:Npn \__clist_item_N_loop:nw #1 #2,
  {
    \int_compare:nNnTF {#1} = 0
      { \__clist_use_i_delimit_by_s_stop:nw { \exp_not:n {#2} } }
      { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
  }
```

```
\cs_generate_variant:Nn \clist_item:Nn { c }
```

(*End of definition for* `\clist_item:Nn`*,* `\__clist_item:nnnN`*, and* `\__clist_item_N_loop:nw`*. This function is documented on page 192.*)

`\clist_item:nn` `\clist_item:en` `\__clist_item_n:nw` `\__clist_item_n_loop:nw` `\__clist_item_n_end:n` `\__clist_item_n_strip:n` `\__clist_item_n_strip:w`

This starts in the same way as `\clist_item:Nn` by counting the items of the comma list. The final item should be space-trimmed before being brace-stripped, hence we insert a couple of odd-looking `\prg_do_nothing:` to avoid losing braces. Blank items are ignored.

```
\cs_new:Npn \clist_item:nn #1#2
  {
    \__clist_item:ffnN
      { \clist_count:n {#1} }
      { \int_eval:n {#2} }
      {#1}
      \__clist_item_n:nw
  }
\cs_generate_variant:Nn \clist_item:nn { e }
\cs_new:Npn \__clist_item_n:nw #1
  { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
\cs_new:Npn \__clist_item_n_loop:nw #1 #2,
  {
    \exp_args:No \tl_if_blank:nTF {#2}
      { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
      {
        \int_compare:nNnTF {#1} = 0
          { \exp_args:No \__clist_item_n_end:n {#2} }
          {
            \exp_args:Nf \__clist_item_n_loop:nw
              { \int_eval:n { #1 - 1 } }
              \prg_do_nothing:
          }
      }
  }
\cs_new:Npn \__clist_item_n_end:n #1 #2 \s__clist_stop
  { \tl_trim_spaces_apply:nN {#1} \__clist_item_n_strip:n }
\cs_new:Npn \__clist_item_n_strip:n #1 { \__clist_item_n_strip:w #1 , }
\cs_new:Npn \__clist_item_n_strip:w #1 , { \exp_not:n {#1} }
```

(*End of definition for* `\clist_item:nn` *and others. This function is documented on page 192.*)

`\clist_rand_item:n` `\clist_rand_item:N` `\clist_rand_item:c` `\__clist_rand_item:nn`

The `N`-type function is not implemented through the `n`-type function for efficiency: for instance comma-list variables do not require space-trimming of their items. Even testing for emptyness of an `n`-type comma-list is slow, so we count items first and use that both for the emptyness test and the pseudo-random integer. Importantly, `\clist_item:Nn` and `\clist_item:nn` only evaluate their argument once.

```
\cs_new:Npn \clist_rand_item:n #1
  { \exp_args:Nf \__clist_rand_item:nn { \clist_count:n {#1} } {#1} }
\cs_new:Npn \__clist_rand_item:nn #1#2
  {
    \int_compare:nNnF {#1} = 0
      { \clist_item:nn {#2} { \int_rand:nn { 1 } {#1} } }
  }
\cs_new:Npn \clist_rand_item:N #1
  {
```

```
    \clist_if_empty:NF #1
      { \clist_item:Nn #1 { \int_rand:nn { 1 } { \clist_count:N #1 } } }
  }
\cs_generate_variant:Nn \clist_rand_item:N { c }
```

(*End of definition for* `\clist_rand_item:n`*,* `\clist_rand_item:N`*, and* `\__clist_rand_item:nn`*. These functions are documented on page 192.*)

61.10 Viewing comma lists

`\clist_show:N`
`\clist_show:c`
`\clist_log:N`
`\clist_log:c`
`\__clist_show:NN`

Apply the general `\__kernel_chk_tl_type:NnnT` with `\exp_not:o #2` serving as a dummy code to prevent a check performed by this auxiliary.

```
\cs_new_protected:Npn \clist_show:N { \__clist_show:NN \msg_show:nneeee }
\cs_generate_variant:Nn \clist_show:N { c }
\cs_new_protected:Npn \clist_log:N { \__clist_show:NN \msg_log:nneeee }
\cs_generate_variant:Nn \clist_log:N { c }
\cs_new_protected:Npn \__clist_show:NN #1#2
  {
    \__kernel_chk_tl_type:NnnT #2 { clist } { \exp_not:o #2 }
      {
        \int_compare:nNnTF { \clist_count:N #2 }
          = { \exp_args:No \clist_count:n #2 }
          {
            #1 { clist } { show }
              { \token_to_str:N #2 }
              { \clist_map_function:NN #2 \msg_show_item:n }
              { } { }
          }
          {
            \msg_error:nnee { clist } { non-clist }
              { \token_to_str:N #2 } { \tl_to_str:N #2 }
          }
      }
  }
```

(*End of definition for* `\clist_show:N`*,* `\clist_log:N`*, and* `\__clist_show:NN`*. These functions are documented on page 192.*)

`\clist_show:n`
`\clist_log:n`
`\__clist_show:Nn`

A variant of the above: no existence check, empty first argument for the message.

```
\cs_new_protected:Npn \clist_show:n { \__clist_show:Nn \msg_show:nneeee }
\cs_new_protected:Npn \clist_log:n { \__clist_show:Nn \msg_log:nneeee }
\cs_new_protected:Npn \__clist_show:Nn #1#2
  {
    #1 { clist } { show }
      { } { \clist_map_function:nN {#2} \msg_show_item:n } { } { }
  }
```

(*End of definition for* `\clist_show:n`*,* `\clist_log:n`*, and* `\__clist_show:Nn`*. These functions are documented on page 193.*)

61.11 Scratch comma lists

\l_tmpa_clist
\l_tmpb_clist
\g_tmpa_clist
\g_tmpb_clist

Temporary comma list variables.

```
\clist_new:N \l_tmpa_clist
\clist_new:N \l_tmpb_clist
\clist_new:N \g_tmpa_clist
\clist_new:N \g_tmpb_clist
```

(*End of definition for* `\l_tmpa_clist` *and others. These variables are documented on page 193.*)

```
⟨/package⟩
```

Chapter 62

l3token implementation

```
⟨*package⟩
⟨*tex⟩
⟨@@=char⟩
```

62.1 Internal auxiliaries

\s__char_stop

Internal scan mark.

```
\scan_new:N \s__char_stop
```

(*End of definition for* \s__char_stop*.*)

\q__char_no_value

Internal recursion quarks.

```
\quark_new:N \q__char_no_value
```

(*End of definition for* \q__char_no_value*.*)

__char_quark_if_no_value_p:N
__char_quark_if_no_value:NTF

Functions to query recursion quarks.

```
\__kernel_quark_new_conditional:Nn \__char_quark_if_no_value:N { TF }
```

(*End of definition for* __char_quark_if_no_value:NTF*.*)

62.2 Manipulating and interrogating character tokens

\char_set_catcode:nn
\char_value_catcode:n
\char_show_value_catcode:n

Simple wrappers around the primitives.

```
\cs_new_protected:Npn \char_set_catcode:nn #1#2
  { \tex_catcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
\cs_new:Npn \char_value_catcode:n #1
  { \tex_the:D \tex_catcode:D \int_eval:n {#1} \exp_stop_f: }
\cs_new_protected:Npn \char_show_value_catcode:n #1
  { \exp_args:Nf \tl_show:n { \char_value_catcode:n {#1} } }
```

(*End of definition for* \char_set_catcode:nn *,* \char_value_catcode:n *, and* \char_show_value_catcode:n*. These functions are documented on page 197.*)

\char_set_catcode_escape:N
\char_set_catcode_group_begin:N
\char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N
\char_set_catcode_alignment:N
\char_set_catcode_end_line:N
\char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
\char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N

```
\cs_new_protected:Npn \char_set_catcode_escape:N #1
  { \char_set_catcode:nn { `#1 } { 0 } }
\cs_new_protected:Npn \char_set_catcode_group_begin:N #1
  { \char_set_catcode:nn { `#1 } { 1 } }
\cs_new_protected:Npn \char_set_catcode_group_end:N #1
  { \char_set_catcode:nn { `#1 } { 2 } }
\cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
  { \char_set_catcode:nn { `#1 } { 3 } }
\cs_new_protected:Npn \char_set_catcode_alignment:N #1
  { \char_set_catcode:nn { `#1 } { 4 } }
\cs_new_protected:Npn \char_set_catcode_end_line:N #1
  { \char_set_catcode:nn { `#1 } { 5 } }
\cs_new_protected:Npn \char_set_catcode_parameter:N #1
  { \char_set_catcode:nn { `#1 } { 6 } }
\cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
  { \char_set_catcode:nn { `#1 } { 7 } }
\cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
  { \char_set_catcode:nn { `#1 } { 8 } }
\cs_new_protected:Npn \char_set_catcode_ignore:N #1
  { \char_set_catcode:nn { `#1 } { 9 } }
\cs_new_protected:Npn \char_set_catcode_space:N #1
  { \char_set_catcode:nn { `#1 } { 10 } }
\cs_new_protected:Npn \char_set_catcode_letter:N #1
  { \char_set_catcode:nn { `#1 } { 11 } }
\cs_new_protected:Npn \char_set_catcode_other:N #1
  { \char_set_catcode:nn { `#1 } { 12 } }
\cs_new_protected:Npn \char_set_catcode_active:N #1
  { \char_set_catcode:nn { `#1 } { 13 } }
\cs_new_protected:Npn \char_set_catcode_comment:N #1
  { \char_set_catcode:nn { `#1 } { 14 } }
\cs_new_protected:Npn \char_set_catcode_invalid:N #1
  { \char_set_catcode:nn { `#1 } { 15 } }
```

(*End of definition for* \char_set_catcode_escape:N *and others. These functions are documented on page 196.*)

\char_set_catcode_escape:n
\char_set_catcode_group_begin:n
\char_set_catcode_group_end:n
\char_set_catcode_math_toggle:n
\char_set_catcode_alignment:n
\char_set_catcode_end_line:n
\char_set_catcode_parameter:n
\char_set_catcode_math_superscript:n
\char_set_catcode_math_subscript:n
\char_set_catcode_ignore:n
\char_set_catcode_space:n
\char_set_catcode_letter:n
\char_set_catcode_other:n
\char_set_catcode_active:n
\char_set_catcode_comment:n
\char_set_catcode_invalid:n

```
\cs_new_protected:Npn \char_set_catcode_escape:n #1
  { \char_set_catcode:nn {#1} { 0 } }
\cs_new_protected:Npn \char_set_catcode_group_begin:n #1
  { \char_set_catcode:nn {#1} { 1 } }
\cs_new_protected:Npn \char_set_catcode_group_end:n #1
  { \char_set_catcode:nn {#1} { 2 } }
\cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
  { \char_set_catcode:nn {#1} { 3 } }
\cs_new_protected:Npn \char_set_catcode_alignment:n #1
  { \char_set_catcode:nn {#1} { 4 } }
\cs_new_protected:Npn \char_set_catcode_end_line:n #1
  { \char_set_catcode:nn {#1} { 5 } }
\cs_new_protected:Npn \char_set_catcode_parameter:n #1
  { \char_set_catcode:nn {#1} { 6 } }
\cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
  { \char_set_catcode:nn {#1} { 7 } }
```

```
\cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
  { \char_set_catcode:nn {#1} { 8 } }
\cs_new_protected:Npn \char_set_catcode_ignore:n #1
  { \char_set_catcode:nn {#1} { 9 } }
\cs_new_protected:Npn \char_set_catcode_space:n #1
  { \char_set_catcode:nn {#1} { 10 } }
\cs_new_protected:Npn \char_set_catcode_letter:n #1
  { \char_set_catcode:nn {#1} { 11 } }
\cs_new_protected:Npn \char_set_catcode_other:n #1
  { \char_set_catcode:nn {#1} { 12 } }
\cs_new_protected:Npn \char_set_catcode_active:n #1
  { \char_set_catcode:nn {#1} { 13 } }
\cs_new_protected:Npn \char_set_catcode_comment:n #1
  { \char_set_catcode:nn {#1} { 14 } }
\cs_new_protected:Npn \char_set_catcode_invalid:n #1
  { \char_set_catcode:nn {#1} { 15 } }
```

(*End of definition for* `\char_set_catcode_escape:n` *and others. These functions are documented on page 197.*)

`\char_set_mathcode:nn`
`\char_value_mathcode:n`
`\char_show_value_mathcode:n`
`\char_set_lccode:nn`
`\char_value_lccode:n`
`\char_show_value_lccode:n`
`\char_set_uccode:nn`
`\char_value_uccode:n`
`\char_show_value_uccode:n`
`\char_set_sfcode:nn`
`\char_value_sfcode:n`
`\char_show_value_sfcode:n`

Pretty repetitive, but necessary!

```
\cs_new_protected:Npn \char_set_mathcode:nn #1#2
  { \tex_mathcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
\cs_new:Npn \char_value_mathcode:n #1
  { \tex_the:D \tex_mathcode:D \int_eval:n {#1} \exp_stop_f: }
\cs_new_protected:Npn \char_show_value_mathcode:n #1
  { \exp_args:Nf \tl_show:n { \char_value_mathcode:n {#1} } }
\cs_new_protected:Npn \char_set_lccode:nn #1#2
  { \tex_lccode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
\cs_new:Npn \char_value_lccode:n #1
  { \tex_the:D \tex_lccode:D \int_eval:n {#1} \exp_stop_f: }
\cs_new_protected:Npn \char_show_value_lccode:n #1
  { \exp_args:Nf \tl_show:n { \char_value_lccode:n {#1} } }
\cs_new_protected:Npn \char_set_uccode:nn #1#2
  { \tex_uccode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
\cs_new:Npn \char_value_uccode:n #1
  { \tex_the:D \tex_uccode:D \int_eval:n {#1} \exp_stop_f: }
\cs_new_protected:Npn \char_show_value_uccode:n #1
  { \exp_args:Nf \tl_show:n { \char_value_uccode:n {#1} } }
\cs_new_protected:Npn \char_set_sfcode:nn #1#2
  { \tex_sfcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
\cs_new:Npn \char_value_sfcode:n #1
  { \tex_the:D \tex_sfcode:D \int_eval:n {#1} \exp_stop_f: }
\cs_new_protected:Npn \char_show_value_sfcode:n #1
  { \exp_args:Nf \tl_show:n { \char_value_sfcode:n {#1} } }
```

(*End of definition for* `\char_set_mathcode:nn` *and others. These functions are documented on page 198.*)

`\l_char_active_seq`
`\l_char_special_seq`

Two sequences for dealing with special characters. The first is characters which may be active, the second longer list is for "special" characters more generally. Both lists are escaped so that for example bulk code assignments can be carried out. In both cases, the order is by ASCII character code (as is done in for example `\ExplSyntaxOn`).

```
\seq_new:N \l_char_special_seq
\seq_set_split:Nnn \l_char_special_seq { }
```

```
    { \  \" \# \$ \% \& \\ \^ \_ \{ \} \~ }
\seq_new:N \l_char_active_seq
\seq_set_split:Nnn \l_char_active_seq { }
    { \" \$ \& \^ \_ \~ }
```

(*End of definition for* `\l_char_active_seq` *and* `\l_char_special_seq`. *These variables are documented on page* *199*.)

62.3 Creating character tokens

`\char_set_active_eq:NN`
`\char_set_active_eq:Nc`
`\char_gset_active_eq:NN`
`\char_gset_active_eq:Nc`
`\char_set_active_eq:nN`
`\char_set_active_eq:nc`
`\char_gset_active_eq:nN`
`\char_gset_active_eq:nc`

Four simple functions with very similar definitions, so set up using an auxiliary. These are similar to LuaTEX's `\letcharcode` primitive.

```
\group_begin:
  \char_set_catcode_active:N \^^@
  \cs_set_protected:Npn \__char_tmp:nN #1#2
    {
      \cs_new_protected:cpn { #1 :nN } ##1
        {
          \group_begin:
            \char_set_lccode:nn { `\^^@ } { ##1 }
          \tex_lowercase:D { \group_end: #2 ^^@ }
        }
      \cs_new_protected:cpe { #1 :NN } ##1
        { \exp_not:c { #1 : nN } { `##1 } }
    }
  \__char_tmp:nN { char_set_active_eq }  \cs_set_eq:NN
  \__char_tmp:nN { char_gset_active_eq } \cs_gset_eq:NN
\group_end:
\cs_generate_variant:Nn \char_set_active_eq:NN  { Nc }
\cs_generate_variant:Nn \char_gset_active_eq:NN { Nc }
\cs_generate_variant:Nn \char_set_active_eq:nN  { nc }
\cs_generate_variant:Nn \char_gset_active_eq:nN { nc }
```

(*End of definition for* `\char_set_active_eq:NN` *and others. These functions are documented on page* *195*.)

`\__char_int_to_roman:w`

For efficiency in 8-bit engines, we use the faster primitive approach to making roman numerals.

```
\cs_new_eq:NN \__char_int_to_roman:w \tex_romannumeral:D
```

(*End of definition for* `\__char_int_to_roman:w`.)

`\char_generate:nn`
`\__char_generate_aux:nn`
`\__char_generate_aux:nnw`
`\__char_generate_auxii:nnw`
`\l__char_tmp_tl`
`\__char_generate_invalid_catcode:`

The aim here is to generate characters of (broadly) arbitrary category code. Where possible, that is done using engine support (XeTEX, LuaTEX). There are though various issues which are covered below. At the interface layer, turn the two arguments into integers up-front so this is only done once.

```
\cs_new:Npn \char_generate:nn #1#2
  {
    \exp:w \exp_after:wN \__char_generate_aux:w
      \int_value:w \int_eval:n {#1} \exp_after:wN ;
      \int_value:w \int_eval:n {#2} ;
  }
```

Before doing any actual conversion, first some special case filtering. Spaces are out here as LuaTEX emulation only makes normal (charcode 32 spaces). However, `^^@` is filtered out separately as that can't be done with macro emulation either, so is treated separately. That done, hand off to the engine-dependent part.

```
\cs_new:Npn \__char_generate_aux:w #1 ; #2 ;
  {
    \if_int_odd:w 0
        \if_int_compare:w #2 < 1  \exp_stop_f: 1 \fi:
        \if_int_compare:w #2 = 5  \exp_stop_f: 1 \fi:
        \if_int_compare:w #2 = 9  \exp_stop_f: 1 \fi:
        \if_int_compare:w #2 > 13 \exp_stop_f: 1 \fi: \exp_stop_f:
      \msg_expandable_error:nn { char }
        { invalid-catcode }
    \else:
      \if_int_odd:w 0
        \if_int_compare:w #1 < \c_zero_int 1 \fi:
        \if_int_compare:w #1 > \c_max_char_int 1 \fi: \exp_stop_f:
        \msg_expandable_error:nn { char }
          { out-of-range }
      \else:
        \if_int_compare:w #2#1 = 100 \exp_stop_f:
          \msg_expandable_error:nn { char } { null-space }
        \else:
          \__char_generate_aux:nnw {#1} {#2}
        \fi:
      \fi:
    \fi:
    \exp_end:
  }
\tl_new:N \l__char_tmp_tl
```

Engine-dependent definitions are now needed for the implementation. Recent (u)pTEX and the Unicode engines LuaTEX and XeTEX have engine-level support for expandable character creation. pdfTEX and older (u)pTEX releases do not. The branching here if low-level to avoid fixing the category code of the null character used in the false branch. The final level is the basic definition at the engine level: the arguments here are integers so there is no need to worry about them too much. Older versions of XeTEX cannot generate active characters so we filter that: at some future stage that may change: the slightly odd ordering of auxiliaries reflects that.

```
\group_begin:
  \char_set_catcode_active:N \^^L
  \cs_set:Npn ^^L { }
  \if_cs_exist:N \tex_Ucharcat:D
      \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
        {
          #3
          \exp_after:wN \exp_end:
          \tex_Ucharcat:D #1 \exp_stop_f: #2 \exp_stop_f:
        }
  \else:
```

For engines where `\Ucharcat` isn't available or emulated, we have to work in macros, and cover only the 8-bit range. The first stage is to build up a `tl` containing `^^@` with each category code that can be accessed in this way, with an error set up for the other cases. This is all done such that it can be quickly accessed using a `\if_case:w` low-level conditional. The list is done in reverse as this puts the case of an active token *first*: that's needed to cover the possibility that it is `\outer`. Getting the braces into the list is done using some standard `\if_false:` manipulation, while all of the `\exp_not:N` are required as there is an expansion in the setup.

```
    \char_set_catcode_active:n { 0 }
    \tl_set:Nn \l__char_tmp_tl { \exp_not:N ^^@ \exp_not:N \or: }
    \char_set_catcode_other:n { 0 }
    \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
    \char_set_catcode_letter:n { 0 }
    \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
```

For making spaces, there needs to be an `o`-type expansion of a `\use:n` (or some other tokenization) to avoid dropping the space.

```
    \tl_put_right:Nn \l__char_tmp_tl { \use:n { ~ } \exp_not:N \or: }
    \tl_put_right:Nn \l__char_tmp_tl { \exp_not:N \or: }
    \char_set_catcode_math_subscript:n { 0 }
    \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
    \char_set_catcode_math_superscript:n { 0 }
    \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
    \char_set_catcode_parameter:n { 0 }
    \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
    \tl_put_right:Nn \l__char_tmp_tl { { \if_false: } \fi: \exp_not:N \or: }
    \char_set_catcode_alignment:n { 0 }
    \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
    \char_set_catcode_math_toggle:n { 0 }
    \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: }
    \char_set_catcode_group_end:n { 0 }
    \tl_put_right:Nn \l__char_tmp_tl { \if_false: { \fi: ^^@ \exp_not:N \or: } % }
    \char_set_catcode_group_begin:n { 0 } % {
    \tl_put_right:Nn \l__char_tmp_tl { ^^@ \exp_not:N \or: } }
```

Convert the above temporary list into a series of constant token lists, one for each character code, using `\tex_lowercase:D` to convert `^^@` in each case. The `e`-type expansion ensures that `\tex_lowercase:D` receives the contents of the token list.

```
      \cs_set_protected:Npn \__char_tmp:n #1
        {
          \char_set_lccode:nn { 0 } {#1}
          \char_set_lccode:nn { 32 } {#1}
          \exp_args:Ne \tex_lowercase:D
            {
              \tl_const:Ne
                \exp_not:c { c__char_ \__char_int_to_roman:w #1 _tl }
                { \exp_not:o \l__char_tmp_tl }
            }
        }
      \int_step_function:nnN { 0 } { 255 }  \__char_tmp:n
```

As TeX is very unhappy if it finds an alignment character inside a primitive `\halign` even when skipping false branches, some precautions are required. TeX is happy if the token is hidden between braces within `\if_false: ... \fi:`. The rather low-level approach here expands in one step to the ⟨*target token*⟩ (`\or:` ...), then `\exp_after:wN` ⟨*target token*⟩ (`\or:` ...) expands in one step to ⟨*target token*⟩. This means that `\exp_not:N` is applied to a potentially-problematic active token.

```
      \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
        {
          #3
          \if_false: { \fi:
          \exp_after:wN \exp_after:wN \exp_after:wN \exp_end:
          \exp_after:wN \exp_after:wN
          \if_case:w \tex_numexpr:D 13 - #2
            \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
            \exp_after:wN \exp_after:wN \exp_after:wN \scan_stop:
            \exp_after:wN \exp_after:wN \exp_after:wN \exp_not:N
              \cs:w c__char_ \__char_int_to_roman:w #1 _tl \cs_end:
          }
          \fi:
        }
  \fi:
\group_end:
```

(*End of definition for* `\char_generate:nn` *and others. This function is documented on page* 195.)

`\c_catcode_active_space_tl`

While `\char_generate:nn` can produce active characters in some engines it cannot in general. It would be possible to simply change the catcode of space but then the code would need to avoid all spaces, making it quite unreadable. Instead we use the primitive `\tex_lowercase:D` trick.

```
\group_begin:
  \char_set_catcode_active:N *
  \char_set_lccode:nn { `* } { `\ }
  \tex_lowercase:D { \tl_const:Nn \c_catcode_active_space_tl { * } }
\group_end:
```

(*End of definition for* `\c_catcode_active_space_tl`*. This variable is documented on page* 195.)

`\c_catcode_other_space_tl`

Create a space with category code 12: an "other" space.

```
\tl_const:Ne \c_catcode_other_space_tl { \char_generate:nn { `\  } { 12 } }
```

(*End of definition for* `\c_catcode_other_space_tl`*. This function is documented on page* 196.)

62.4 Generic tokens

```
⟨@@=token⟩
```

`\s__token_mark`
`\s__token_stop`

Internal scan marks.

```
\scan_new:N \s__token_mark
\scan_new:N \s__token_stop
```

(*End of definition for* `\s__token_mark` *and* `\s__token_stop`*.*)

`\token_to_meaning:N`
`\token_to_meaning:c`
`\token_to_str:N`
`\token_to_str:c`

These are all defined in l3basics, as they are needed "early". This is just a reminder!

(*End of definition for* `\token_to_meaning:N` *and* `\token_to_str:N`*. These functions are documented on page* 200.)

`\token_to_catcode:N` `\__token_to_catcode:N` The macro works by comparing the input token with `\if_catcode:w` with all valid category codes. Since the most common tokens in an average argument list are of category 11 or 12 those are tested first. And since a space and braces are no ordinary N-type arguments, and only control sequences let to those categories can match them they are tested last.

```
\cs_new:Npn \token_to_catcode:N
  { \int_value:w \group_align_safe_begin: \__token_to_catcode:N }
\cs_new:Npn \__token_to_catcode:N #1
  {
    \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
      11
    \else:
      \if_catcode:w \exp_not:N #1 \c_catcode_other_token
        12
      \else:
        \if_catcode:w \exp_not:N #1 \c_math_toggle_token
          3
        \else:
          \if_catcode:w \exp_not:N #1 \c_alignment_token
            4
          \else:
            \if_catcode:w \exp_not:N #1 ##
              6
            \else:
              \if_catcode:w \exp_not:N #1 \c_math_superscript_token
                7
              \else:
                \if_catcode:w \exp_not:N #1 \c_math_subscript_token
                  8
                \else:
                  \if_catcode:w \exp_not:N #1 \c_group_begin_token
                    1
                  \else:
                    \if_catcode:w \exp_not:N #1 \c_group_end_token
                      2
                    \else:
                      \if_catcode:w \exp_not:N #1 \c_space_token
                        10
                      \else:
                        \token_if_cs:NTF #1 { 16 } { 13 }
                      \fi:
                    \fi:
                  \fi:
                \fi:
              \fi:
            \fi:
          \fi:
        \fi:
      \fi:
    \fi:
    \group_align_safe_end:
    \exp_stop_f:
  }
```

(*End of definition for* `\token_to_catcode:N` *and* `\__token_to_catcode:N`*. This function is documented on page* 200*.*)

`\c_group_begin_token`
`\c_group_end_token`
`\c_math_toggle_token`
`\c_alignment_token`
`\c_parameter_token`
`\c_math_superscript_token`
`\c_math_subscript_token`
`\c_space_token`
`\c_catcode_letter_token`
`\c_catcode_other_token`

We define these useful tokens. For the brace and space tokens things have to be done by hand: the formal argument spec. for `\cs_new_eq:NN` does not cover them so we do things by hand. (As currently coded it would *work* with `\cs_new_eq:NN` but that's not really a great idea to show off: we want people to stick to the defined interfaces and that includes us.) So that these few odd names go into the log when appropriate there is a need to hand-apply the `\__kernel_chk_if_free_cs:N` check.

```
\group_begin:
  \__kernel_chk_if_free_cs:N \c_group_begin_token
  \tex_global:D \tex_let:D \c_group_begin_token {
  \__kernel_chk_if_free_cs:N \c_group_end_token
  \tex_global:D \tex_let:D \c_group_end_token }
  \char_set_catcode_math_toggle:N \*
  \cs_new_eq:NN \c_math_toggle_token *
  \char_set_catcode_alignment:N \*
  \cs_new_eq:NN \c_alignment_token *
  \cs_new_eq:NN \c_parameter_token #
  \cs_new_eq:NN \c_math_superscript_token ^
  \char_set_catcode_math_subscript:N \*
  \cs_new_eq:NN \c_math_subscript_token *
  \__kernel_chk_if_free_cs:N \c_space_token
  \use:n { \tex_global:D \tex_let:D \c_space_token = ~ } ~
  \cs_new_eq:NN \c_catcode_letter_token a
  \cs_new_eq:NN \c_catcode_other_token 1
\group_end:
```

(*End of definition for* `\c_group_begin_token` *and others. These functions are documented on page* 199*.*)

`\c_catcode_active_tl`

Not an implicit token!

```
\group_begin:
  \char_set_catcode_active:N \*
  \tl_const:Nn \c_catcode_active_tl { \exp_not:N * }
\group_end:
```

(*End of definition for* `\c_catcode_active_tl`*. This variable is documented on page* 199*.*)

62.5 Token conditionals

`\token_if_group_begin_p:N`
`\token_if_group_begin:NTF`

Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.

```
\prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_group_begin_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End of definition for* `\token_if_group_begin:NTF`*. This function is documented on page* 200*.*)

`\token_if_group_end_p:N`
`\token_if_group_end:NTF`

Check if token is a end group token. We use the constant `\c_group_end_token` for this.

```
\prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T ,  F , TF }
  {
```

```
    \if_catcode:w \exp_not:N #1 \c_group_end_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End of definition for* `\token_if_group_end:NTF`. *This function is documented on page 200.*)

`\token_if_math_toggle_p:N`
`\token_if_math_toggle:NTF`

Check if token is a math shift token. We use the constant `\c_math_toggle_token` for this.

```
\prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_math_toggle_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End of definition for* `\token_if_math_toggle:NTF`. *This function is documented on page 201.*)

`\token_if_alignment_p:N`
`\token_if_alignment:NTF`

Check if token is an alignment tab token. We use the constant `\c_alignment_token` for this.

```
\prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_alignment_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End of definition for* `\token_if_alignment:NTF`. *This function is documented on page 201.*)

`\token_if_parameter_p:N`
`\token_if_parameter:NTF`

Check if token is a parameter token. We use the constant `\c_parameter_token` for this. We have to trick TeX a bit to avoid an error message: within a group we prevent `\c_parameter_token` from behaving like a macro parameter character. The definitions of `\prg_new_conditional:Npnn` are global, so they remain after the group.

```
\group_begin:
\cs_set_eq:NN \c_parameter_token \scan_stop:
\prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_parameter_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
\group_end:
```

(*End of definition for* `\token_if_parameter:NTF`. *This function is documented on page 201.*)

`\token_if_math_superscript_p:N`
`\token_if_math_superscript:NTF`

Check if token is a math superscript token. We use the constant `\c_math_superscript_token` for this.

```
\prg_new_conditional:Npnn \token_if_math_superscript:N #1
  { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_math_superscript_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End of definition for* `\token_if_math_superscript:NTF`. *This function is documented on page 201.*)

\token_if_math_subscript_p:N
\token_if_math_subscript:NTF

Check if token is a math subscript token. We use the constant \c_math_subscript_token for this.

```
\prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_math_subscript_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End of definition for* \token_if_math_subscript:NTF. *This function is documented on page 201.*)

\token_if_space_p:N
\token_if_space:NTF

Check if token is a space token. We use the constant \c_space_token for this.

```
\prg_new_conditional:Npnn \token_if_space:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_space_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End of definition for* \token_if_space:NTF. *This function is documented on page 201.*)

\token_if_letter_p:N
\token_if_letter:NTF

Check if token is a letter token. We use the constant \c_catcode_letter_token for this.

```
\prg_new_conditional:Npnn \token_if_letter:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End of definition for* \token_if_letter:NTF. *This function is documented on page 201.*)

\token_if_other_p:N
\token_if_other:NTF

Check if token is an other char token. We use the constant \c_catcode_other_token for this.

```
\prg_new_conditional:Npnn \token_if_other:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_catcode_other_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End of definition for* \token_if_other:NTF. *This function is documented on page 201.*)

\token_if_active_p:N
\token_if_active:NTF

Check if token is an active char token. We use the constant \c_catcode_active_tl for this. A technical point is that \c_catcode_active_tl is in fact a macro expanding to \exp_not:N *, where * is active.

```
\prg_new_conditional:Npnn \token_if_active:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End of definition for* \token_if_active:NTF. *This function is documented on page 201.*)

\token_if_eq_meaning_p:NN
\token_if_eq_meaning:NNTF

Check if the tokens #1 and #2 have same meaning.

```
\prg_new_eq_conditional:NNn \token_if_eq_meaning:NN \cs_if_eq:NN
  { p , T , F , TF }
```

(*End of definition for* \token_if_eq_meaning:NNTF. *This function is documented on page 202.*)

`\token_if_eq_catcode_p:NN`
`\token_if_eq_catcode:NNTF`

Check if the tokens `#1` and `#2` have same category code.

```
\prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \exp_not:N #2
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End of definition for* `\token_if_eq_catcode:NNTF`. *This function is documented on page* 201.)

`\token_if_eq_charcode_p:NN`
`\token_if_eq_charcode:NNTF`

Check if the tokens `#1` and `#2` have same character code.

```
\prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T ,  F , TF }
  {
    \if_charcode:w \exp_not:N #1 \exp_not:N #2
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End of definition for* `\token_if_eq_charcode:NNTF`. *This function is documented on page* 202.)

`\token_if_macro_p:N`
`\token_if_macro:NTF`
`\__token_if_macro_p:w`

When a token is a macro, `\token_to_meaning:N` always outputs something like `\long macro:#1->#1` so we could naively check to see if the meaning contains `->`. However, this can fail the five `\...mark` primitives, whose meaning has the form `...mark:`⟨*user material*⟩. The problem is that the ⟨*user material*⟩ can contain `->`.

However, only characters, macros, and marks can contain the colon character. The idea is thus to grab until the first `:`, and analyse what is left. However, macros can have any combination of `\long`, `\protected` or `\outer` (not used in LaTeX3) before the string `macro:`. We thus only select the part of the meaning between the first `ma` and the first following `:`. If this string is `cro`, then we have a macro. If the string is `rk`, then we have a mark. The string can also be `cro parameter character` for a colon with a weird category code (namely the usual category code of `#`). Otherwise, it is empty.

This relies on the fact that `\long`, `\protected`, `\outer` cannot contain `ma`, regardless of the escape character, even if the escape character is `m`...

Both `ma` and `:` must be of category code 12 (other), so are detokenized.

```
\use:e
  {
    \prg_new_conditional:Npnn \exp_not:N \token_if_macro:N #1
      { p , T ,  F , TF }
      {
        \exp_not:N \exp_after:wN \exp_not:N \__token_if_macro_p:w
        \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma : }
          \s__token_stop
      }
    \cs_new:Npn \exp_not:N  \__token_if_macro_p:w
      #1 \tl_to_str:n { ma } #2 \c_colon_str #3 \s__token_stop
  }
      {
        \str_if_eq:nnTF { #2 } { cro }
          { \prg_return_true: }
          { \prg_return_false: }
      }
```

(*End of definition for* `\token_if_macro:NTF` *and* `\__token_if_macro_p:w`. *This function is documented on page* 202.)

\token_if_cs_p:N
\token_if_cs:NTF

Check if token has same catcode as a control sequence. This follows the same pattern as for \token_if_letter:N *etc.* We use \scan_stop: for this.

```
\prg_new_conditional:Npnn \token_if_cs:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \scan_stop:
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End of definition for* \token_if_cs:NTF. *This function is documented on page 202.*)

\token_if_expandable_p:N
\token_if_expandable:NTF

Check if token is expandable. We use the fact that TeX temporarily converts \exp_-not:N ⟨*token*⟩ into \scan_stop: if ⟨*token*⟩ is expandable. An undefined token is not considered as expandable. No problem nesting the conditionals, since the third #1 is only skipped if it is non-expandable (hence not part of TeX's conditional apparatus).

```
\prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T ,  F , TF }
  {
    \exp_after:wN \if_meaning:w \exp_not:N #1 #1
      \prg_return_false:
    \else:
      \if_cs_exist:N #1
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \fi:
  }
```

(*End of definition for* \token_if_expandable:NTF. *This function is documented on page 202.*)

__token_delimit_by_char":w
__token_delimit_by_count:w
__token_delimit_by_dimen:w
__token_delimit_by_␣font:w
__token_delimit_by_macro:w
__token_delimit_by_muskip:w
__token_delimit_by_skip:w
__token_delimit_by_toks:w

These auxiliary functions are used below to define some conditionals which detect whether the \meaning of their argument begins with a particular string. Each auxiliary takes an argument delimited by a string, a second one delimited by \s__token_stop, and returns the first one and its delimiter. This result is eventually compared to another string. Note that the "font" auxiliary is delimited by a space followed by "font". This avoids an unnecessary check for the \font primitive below.

```
\group_begin:
\cs_set_protected:Npn \__token_tmp:w #1
  {
    \use:e
      {
        \cs_new:Npn \exp_not:c { __token_delimit_by_ #1 :w }
            ##1 \tl_to_str:n {#1} ##2 \s__token_stop
          { ##1 \tl_to_str:n {#1} }
      }
  }
\__token_tmp:w { char" }
\__token_tmp:w { count }
\__token_tmp:w { dimen }
\__token_tmp:w { ~ font }
\__token_tmp:w { macro }
\__token_tmp:w { muskip }
\__token_tmp:w { skip }
\__token_tmp:w { toks }
\group_end:
```

(*End of definition for* `\__token_delimit_by_char":w` *and others.*)

`\token_if_chardef_p:N`
`\token_if_chardef:NTF`
`\token_if_mathchardef_p:N`
`\token_if_mathchardef:NTF`
`\token_if_long_macro_p:N`
`\token_if_long_macro:NTF`
`\token_if_protected_macro_p:N`
`\token_if_protected_macro:NTF`
`\token_if_protected_long_macro_p:N`
`\token_if_protected_long_macro:NTF`
`\token_if_font_selection_p:N`
`\token_if_font_selection:NTF`
`\token_if_dim_register_p:N`
`\token_if_dim_register:NTF`
`\token_if_int_register_p:N`
`\token_if_int_register:NTF`
`\token_if_muskip_register_p:N`
`\token_if_muskip_register:NTF`
`\token_if_skip_register_p:N`
`\token_if_skip_register:NTF`
`\token_if_toks_register_p:N`
`\token_if_toks_register:NTF`

Each of these conditionals tests whether its argument's `\meaning` starts with a given string. This is essentially done by having an auxiliary grab an argument delimited by the string and testing whether the argument was empty. Of course, a copy of this string must first be added to the end of the `\meaning` to avoid a runaway argument in case it does not contain the string. Two complications arise. First, the escape character is not fixed, and cannot be included in the delimiter of the auxiliary function (this function cannot be defined on the fly because tests must remain expandable): instead the first argument of the auxiliary (plus the delimiter to avoid complications with trailing spaces) is compared using `\str_if_eq:eeTF` to the result of applying `\token_to_str:N` to a control sequence. Second, the `\meaning` of primitives such as `\dimen` or `\dimendef` starts in the same way as registers such as `\dimen123`, so they must be tested for.

Characters used as delimiters must have catcode 12 and are obtained through `\tl_to_str:n`. This requires doing all definitions within e-expansion. The temporary function `\__token_tmp:w` used to define each conditional receives three arguments: the name of the conditional, the auxiliary's delimiter (also used to name the auxiliary), and the string to which one compares the auxiliary's result. Note that the `\meaning` of a protected long macro starts with `\protected\long macro`, with no space after `\protected` but a space after `\long`, hence the mixture of `\token_to_str:N` and `\tl_to_str:n`.

For the first six conditionals, `\cs_if_exist:cT` turns out to be `false` (thanks to the leading space for `font`), and the code boils down to a string comparison between the result of the auxiliary on the `\meaning` of the conditional's argument `####1`, and `#3`. Both are evaluated at run-time, as this is important to get the correct escape character.

The other five conditionals have additional code that compares the argument `####1` to two TeX primitives which would wrongly be recognized as registers otherwise. Despite using TeX's primitive conditional construction, this does not break when `####1` is itself a conditional, because branches of the conditionals are only skipped if `####1` is one of the two primitives that are tested for (which are not TeX conditionals).

```
\group_begin:
\cs_set_protected:Npn \__token_tmp:w #1#2#3
  {
    \use:e
      {
        \prg_new_conditional:Npnn \exp_not:c { token_if_ #1 :N } ##1
          { p , T ,  F , TF }
          {
            \cs_if_exist:cT { tex_ #2 :D }
              {
                \exp_not:N \if_meaning:w ##1 \exp_not:c { tex_ #2 :D }
                \exp_not:N \prg_return_false:
                \exp_not:N \else:
                \exp_not:N \if_meaning:w ##1 \exp_not:c { tex_ #2 def:D }
                \exp_not:N \prg_return_false:
                \exp_not:N \else:
              }
            \exp_not:N \str_if_eq:eeTF
              {
                \exp_not:N \exp_after:wN
                \exp_not:c { __token_delimit_by_ #2 :w }
                \exp_not:N \token_to_meaning:N ##1
                ? \tl_to_str:n {#2} \s__token_stop
```

```
                }
                { \exp_not:n {#3} }
                { \exp_not:N \prg_return_true: }
                { \exp_not:N \prg_return_false: }
              \cs_if_exist:cT { tex_ #2 :D }
                {
                  \exp_not:N \fi:
                  \exp_not:N \fi:
                }
            }
        }
    }
\__token_tmp:w { chardef } { char" } { \token_to_str:N \char" }
\__token_tmp:w { mathchardef } { char" } { \token_to_str:N \mathchar" }
\__token_tmp:w { long_macro } { macro } { \tl_to_str:n { \long } macro }
\__token_tmp:w { protected_macro } { macro }
  { \tl_to_str:n { \protected } macro }
\__token_tmp:w { protected_long_macro } { macro }
  { \token_to_str:N \protected \tl_to_str:n { \long } macro }
\__token_tmp:w { font_selection } { ~ font } { select ~ font }
\__token_tmp:w { dim_register } { dimen } { \token_to_str:N \dimen }
\__token_tmp:w { int_register } { count } { \token_to_str:N \count }
\__token_tmp:w { muskip_register } { muskip } { \token_to_str:N \muskip }
\__token_tmp:w { skip_register } { skip } { \token_to_str:N \skip }
\__token_tmp:w { toks_register } { toks } { \token_to_str:N \toks }
\group_end:
```

(*End of definition for* `\token_if_chardef:NTF` *and others. These functions are documented on page* *202*.)

`\token_if_primitive_p:N`
`\token_if_primitive:NTF`
`\__token_if_primitive:NNw`
`\__token_if_primitive_space:w`
`\__token_if_primitive_nullfont:N`
`\__token_if_primitive_loop:N`
`\__token_if_primitive:Nw`
`\__token_if_primitive_undefined:N`
`\__token_if_primitive_lua:N`

We filter out macros first, because they cause endless trouble later otherwise.

Primitives are almost distinguished by the fact that the result of `\token_to_meaning:N` is formed from letters only. Every other token has either a space (e.g., `the letter A`), a digit (e.g., `\count123`) or a double quote (e.g., `\char"A`).

Ten exceptions: on the one hand, `\tex_undefined:D` is not a primitive, but its meaning is `undefined`, only letters; on the other hand, `\space`, `\italiccorr`, `\hyphen`, `\firstmark`, `\topmark`, `\botmark`, `\splitfirstmark`, `\splitbotmark`, and `\nullfont` are primitives, but have non-letters in their meaning.

We start by removing the two first (non-space) characters from the meaning. This removes the escape character (which may be nonexistent depending on `\endlinechar`), and takes care of three of the exceptions: `\space`, `\italiccorr` and `\hyphen`, whose meaning is at most two characters. This leaves a string terminated by some `:`, and `\s__token_stop`.

The meaning of each one of the five `\...mark` primitives has the form ⟨*letters*⟩`:`⟨*user material*⟩. In other words, the first non-letter is a colon. We remove everything after the first colon.

We are now left with a string, which we must analyze. For primitives, it contains only letters. For non-primitives, it contains either `"`, or a space, or a digit. Two exceptions remain: `\tex_undefined:D`, which is not a primitive, and `\nullfont`, which is a primitive.

Spaces cannot be grabbed in an undelimited way, so we check them separately. If there is a space, we test for `\nullfont`. Otherwise, we go through characters one by one, and stop at the first character less than `` `A `` (this is not quite a test for "only letters", but is close enough to work in this context). If this first character is : then we have a primitive, or `\tex_undefined:D`, and if it is " or a digit, then the token is not a primitive.

For LuaTEX we use a different implementation which just looks at the command code for the token and compares it to a list of non-primitives. Again, `\nullfont` is a special case because it is the only primitive with the normally non-primitive `set_font` command code.

In LuaMetaTEX some of the command names are different, so we check for both versions. The first one is always the LuaTEX version.

```
\sys_if_engine_luatex:TF
  {
⟨/tex⟩
⟨*lua⟩
do
  local get_next = token.get_next
  local get_command = token.get_command
  local get_index = token.get_index
  local get_mode = token.get_mode or token.get_index
  local cmd = command_id
  local set_font = cmd'get_font'
  local biggest_char = token.biggest_char and token.biggest_char()
                    or status.getconstants().max_character_code

  local mode_below_biggest_char = {}
  local index_not_nil = {}
  local mode_not_null = {}
  local non_primitive = {
    [cmd'left_brace'] = true,
    [cmd'right_brace'] = true,
    [cmd'math_shift'] = true,
    [cmd'mac_param' or cmd'parameter'] = mode_below_biggest_char,
    [cmd'sup_mark' or cmd'superscript'] = true,
    [cmd'sub_mark' or cmd'subscript'] = true,
    [cmd'endv' or cmd'ignore'] = true,
    [cmd'spacer'] = true,
    [cmd'letter'] = true,
    [cmd'other_char'] = true,
    [cmd'tab_mark' or cmd'alignment_tab'] = mode_below_biggest_char,
    [cmd'char_given'] = true,
    [cmd'math_given' or 'math_char_given'] = true,
    [cmd'xmath_given' or 'math_char_xgiven'] = true,
    [cmd'set_font'] = mode_not_null,
    [cmd'undefined_cs'] = true,
    [cmd'call'] = true,
    [cmd'long_call' or cmd'protected_call'] = true,
    [cmd'outer_call' or cmd'tolerant_call'] = true,
    [cmd'long_outer_call' or cmd'tolerant_protected_call'] = true,
    [cmd'assign_glue' or cmd'register_glue'] = index_not_nil,
    [cmd'assign_mu_glue' or cmd'register_mu_glue' or cmd'register_muglue'] = index_not_nil,
    [cmd'assign_toks' or cmd'register_toks'] = index_not_nil,
    [cmd'assign_int' or cmd'register_int' or cmd'register_integer'] = index_not_nil,
    [cmd'assign_attr' or cmd'register_attribute'] = true,
    [cmd'assign_dimen' or cmd'register_dimen' or cmd'register_dimension'] = index_not_nil,
  }
```

```
  luacmd("__token_if_primitive_lua:N", function()
    local tok = get_next()
    local is_non_primitive = non_primitive[get_command(tok)]
    return put_next(
           is_non_primitive == true
             and false_tok
        or is_non_primitive == nil
             and true_tok
        or is_non_primitive == mode_not_null
             and (get_mode(tok) == 0 and true_tok or false_tok)
        or is_non_primitive == index_not_nil
             and (get_index(tok) and false_tok or true_tok)
        or is_non_primitive == mode_below_biggest_char
             and (get_mode(tok) > biggest_char and true_tok or false_tok))
  end, "global")
end
⟨/lua⟩
⟨*tex⟩
    \prg_new_conditional:Npnn \token_if_primitive:N #1 { p , T , F , TF }
      {
        \__token_if_primitive_lua:N #1
      }
  }
  {
    \tex_global:D \tex_chardef:D \c__token_A_int = `A ~ %
    \use:e
      {
        \prg_new_conditional:Npnn \exp_not:N \token_if_primitive:N #1
          { p , T , F , TF }
          {
            \exp_not:N \token_if_macro:NTF #1
              \exp_not:N \prg_return_false:
              {
                \exp_not:N \exp_after:wN \exp_not:N \__token_if_primitive:NNw
                \exp_not:N \token_to_meaning:N #1
                  \tl_to_str:n { : : : } \s__token_stop #1
              }
          }
        \cs_new:Npn \exp_not:N \__token_if_primitive:NNw
          #1#2 #3 \c_colon_str #4 \s__token_stop
          {
            \exp_not:N \tl_if_empty:oTF
              { \exp_not:N \__token_if_primitive_space:w #3 ~ }
              {
                \exp_not:N \__token_if_primitive_loop:N #3
                  \c_colon_str \s__token_stop
              }
              { \exp_not:N \__token_if_primitive_nullfont:N }
          }
      }
    \cs_new:Npn \__token_if_primitive_space:w #1 ~ { }
    \cs_new:Npn \__token_if_primitive_nullfont:N #1
      {
```

```
      \if_meaning:w \tex_nullfont:D #1
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    }
  \cs_new:Npn \__token_if_primitive_loop:N #1
    {
      \if_int_compare:w `#1 < \c__token_A_int %
        \exp_after:wN \__token_if_primitive:Nw
        \exp_after:wN #1
      \else:
        \exp_after:wN \__token_if_primitive_loop:N
      \fi:
    }
  \cs_new:Npn \__token_if_primitive:Nw #1 #2 \s__token_stop
    {
      \if:w : #1
        \exp_after:wN \__token_if_primitive_undefined:N
      \else:
        \prg_return_false:
        \exp_after:wN \use_none:n
      \fi:
    }
  \cs_new:Npn \__token_if_primitive_undefined:N #1
    {
      \if_cs_exist:N #1
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    }
}
```

(*End of definition for* `\token_if_primitive:NTF` *and others. This function is documented on page* *203*.)

`\token_case_catcode:Nn`
`\token_case_catcode:NnTF`
`\token_case_charcode:Nn`
`\token_case_charcode:NnTF`
`\token_case_meaning:Nn`
`\token_case_meaning:NnTF`
`\__token_case:NNnTF`
`\__token_case:NNw`
`\__token_case_end:nw`

The aim here is to allow the case statement to be evaluated using a known number of expansion steps (two), and without needing to use an explicit "end of recursion" marker. That is achieved by using the test input as the final case, as this is always true. The trick is then to tidy up the output such that the appropriate case code plus either the `true` or `false` branch code is inserted.

```
\cs_new:Npn \token_case_catcode:Nn #1#2
  { \exp:w \__token_case:NNnTF \token_if_eq_catcode:NNTF #1 {#2} { } { } }
\cs_new:Npn \token_case_catcode:NnT #1#2#3
  { \exp:w \__token_case:NNnTF \token_if_eq_catcode:NNTF #1 {#2} {#3} { } }
\cs_new:Npn \token_case_catcode:NnF #1#2
  { \exp:w \__token_case:NNnTF \token_if_eq_catcode:NNTF #1 {#2} { } }
\cs_new:Npn \token_case_catcode:NnTF
  { \exp:w \__token_case:NNnTF \token_if_eq_catcode:NNTF }
\cs_new:Npn \token_case_charcode:Nn #1#2
  { \exp:w \__token_case:NNnTF \token_if_eq_charcode:NNTF #1 {#2} { } { } }
\cs_new:Npn \token_case_charcode:NnT #1#2#3
  { \exp:w \__token_case:NNnTF \token_if_eq_charcode:NNTF #1 {#2} {#3} { } }
\cs_new:Npn \token_case_charcode:NnF #1#2
```

```
  { \exp:w \__token_case:NNnTF \token_if_eq_charcode:NNTF #1 {#2} { } }
\cs_new:Npn \token_case_charcode:NnTF
  { \exp:w \__token_case:NNnTF \token_if_eq_charcode:NNTF }
\cs_new:Npn \token_case_meaning:Nn #1#2
  { \exp:w \__token_case:NNnTF \token_if_eq_meaning:NNTF #1 {#2} { } { } }
\cs_new:Npn \token_case_meaning:NnT #1#2#3
  { \exp:w \__token_case:NNnTF \token_if_eq_meaning:NNTF #1 {#2} {#3} { } }
\cs_new:Npn \token_case_meaning:NnF #1#2
  { \exp:w \__token_case:NNnTF \token_if_eq_meaning:NNTF #1 {#2} { } }
\cs_new:Npn \token_case_meaning:NnTF
  { \exp:w \__token_case:NNnTF \token_if_eq_meaning:NNTF }
\cs_new:Npn \__token_case:NNnTF #1#2#3#4#5
  {
    \__token_case:NNw #1 #2 #3 #2 { }
    \s__token_mark {#4}
    \s__token_mark {#5}
    \s__token_stop
  }
\cs_new:Npn \__token_case:NNw #1#2#3#4
  {
    #1 #2 #3
      { \__token_case_end:nw {#4} }
      { \__token_case:NNw #1 #2 }
  }
```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then `#1` is the code to insert, `#2` is the *next* case to check on and `#3` is all of the rest of the cases code. That means that `#4` is the `true` branch code, and `#5` tidies up the spare `\s__token_mark` and the `false` branch. On the other hand, if none of the cases matched then we arrive here using the "termination" case of comparing the search with itself. That means that `#1` is empty, `#2` is the first `\s__token_mark` and so `#4` is the `false` code (the `true` code is mopped up by `#3`).

```
\cs_new:Npn \__token_case_end:nw #1#2#3 \s__token_mark #4#5 \s__token_stop
  { \exp_end: #1 #4 }
```

(*End of definition for* `\token_case_catcode:NnTF` *and others. These functions are documented on page 204.*)

62.6 Peeking ahead at the next token

```
⟨@@=peek⟩
```

Peeking ahead is implemented using a two part mechanism. The outer level provides a defined interface to the lower level material. This allows a large amount of code to be shared. There are four cases:

1. peek at the next token;
2. peek at the next non-space token;
3. peek at the next token and remove it;
4. peek at the next non-space token and remove it.

\l_peek_token
\g_peek_token

Storage tokens which are publicly documented: the token peeked.

```
\cs_new_eq:NN \l_peek_token ?
\cs_new_eq:NN \g_peek_token ?
```

(*End of definition for* `\l_peek_token` *and* `\g_peek_token`*. These variables are documented on page 204.*)

\l__peek_search_token

The token to search for as an implicit token: *cf.* `\l__peek_search_tl`.

```
\cs_new_eq:NN \l__peek_search_token ?
```

(*End of definition for* `\l__peek_search_token`*.*)

\l__peek_search_tl

The token to search for as an explicit token: *cf.* `\l__peek_search_token`.

```
\tl_new:N \l__peek_search_tl
```

(*End of definition for* `\l__peek_search_tl`*.*)

__peek_true:w
__peek_true_aux:w
__peek_false:w
__peek_tmp:w

Functions used by the branching and space-stripping code.

```
\cs_new:Npn \__peek_true:w  { }
\cs_new:Npn \__peek_true_aux:w  { }
\cs_new:Npn \__peek_false:w { }
\cs_new:Npn \__peek_tmp:w { }
```

(*End of definition for* `\__peek_true:w` *and others.*)

\s__peek_mark
\s__peek_stop

Internal scan marks.

```
\scan_new:N \s__peek_mark
\scan_new:N \s__peek_stop
```

(*End of definition for* `\s__peek_mark` *and* `\s__peek_stop`*.*)

__peek_use_none_delimit_by_s_stop:w

Functions to gobble up to a scan mark.

```
\cs_new:Npn \__peek_use_none_delimit_by_s_stop:w #1 \s__peek_stop { }
```

(*End of definition for* `\__peek_use_none_delimit_by_s_stop:w`*.*)

\peek_after:Nw
\peek_gafter:Nw

Simple wrappers for `\futurelet`: no arguments absorbed here.

```
\cs_new_protected:Npn \peek_after:Nw
  { \tex_futurelet:D \l_peek_token }
\cs_new_protected:Npn \peek_gafter:Nw
  { \tex_global:D \tex_futurelet:D \g_peek_token }
```

(*End of definition for* `\peek_after:Nw` *and* `\peek_gafter:Nw`*. These functions are documented on page 204.*)

__peek_true_remove:w

A function to remove the next token and then regain control.

```
\cs_new_protected:Npn \__peek_true_remove:w
  {
    \tex_afterassignment:D \__peek_true_aux:w
    \cs_set_eq:NN \__peek_tmp:w
  }
```

(*End of definition for* `\__peek_true_remove:w`*.*)

\peek_remove_spaces:n
__peek_remove_spaces:

Repeatedly use __peek_true_remove:w to remove a space and call __peek_true_aux:w.

```
\cs_new_protected:Npn \peek_remove_spaces:n #1
  {
    \cs_set:Npe \__peek_false:w { \exp_not:n {#1} }
    \group_align_safe_begin:
    \cs_set:Npn \__peek_true_aux:w { \peek_after:Nw \__peek_remove_spaces: }
    \__peek_true_aux:w
  }
\cs_new_protected:Npn \__peek_remove_spaces:
  {
    \if_meaning:w \l_peek_token \c_space_token
      \exp_after:wN \__peek_true_remove:w
    \else:
      \group_align_safe_end:
      \exp_after:wN \__peek_false:w
    \fi:
  }
```

(*End of definition for* \peek_remove_spaces:n *and* __peek_remove_spaces:. *This function is documented on page 205.*)

\peek_remove_filler:n
__peek_remove_filler:w
__peek_remove_filler:
__peek_remove_filler_expand:w

Here we expand the input, removing spaces and \scan_stop: tokens until we reach a non-expandable token. At that stage we re-insert the payload. To deal with the problem of & tokens, we have to put the align-safe group in the correct place.

```
\cs_new_protected:Npn \peek_remove_filler:n #1
  {
    \cs_set:Npn \__peek_true_aux:w { \__peek_remove_filler:w }
    \cs_set:Npe \__peek_false:w
      {
        \exp_not:N \group_align_safe_end:
        \exp_not:n {#1}
      }
    \group_align_safe_begin:
    \__peek_remove_filler:w
  }
\cs_new_protected:Npn \__peek_remove_filler:w
  {
    \exp_after:wN \peek_after:Nw \exp_after:wN \__peek_remove_filler:
    \exp:w \exp_end_continue_f:w
  }
```

Here we can nest conditionals as \l_peek_token is only skipped over in the nested one if it's a space: no problems with conditionals or outer tokens.

```
\cs_new_protected:Npn \__peek_remove_filler:
  {
    \if_catcode:w \exp_not:N \l_peek_token \c_space_token
      \exp_after:wN \__peek_true_remove:w
    \else:
      \if_meaning:w \l_peek_token \scan_stop:
        \exp_after:wN \exp_after:wN \exp_after:wN
          \__peek_true_remove:w
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN
```

```
        \__peek_remove_filler_expand:w
      \fi:
    \fi:
  }
```

To deal with undefined control sequences in the same way TeX does, we need to check for expansion manually.

```
\cs_new_protected:Npn \__peek_remove_filler_expand:w
  {
    \exp_after:wN \if_meaning:w \exp_not:N \l_peek_token \l_peek_token
      \exp_after:wN \__peek_false:w
    \else:
      \exp_after:wN \__peek_remove_filler:w
    \fi:
  }
```

(*End of definition for* `\peek_remove_filler:n` *and others. This function is documented on page 206.*)

`\__peek_token_generic_aux:NNNTF`

The generic functions store the test token in both implicit and explicit modes, and the `true` and `false` code as token lists, more or less. The two branches have to be absorbed here as the input stream needs to be cleared for the peek function itself. Here, `#1` is `\__peek_true_remove:w` when removing the token and `\__peek_true_aux:w` otherwise.

```
\cs_new_protected:Npn \__peek_token_generic_aux:NNNTF #1#2#3#4#5
  {
    \group_align_safe_begin:
    \cs_set_eq:NN \l__peek_search_token #3
    \tl_set:Nn \l__peek_search_tl {#3}
    \cs_set:Npe \__peek_true_aux:w
      {
        \exp_not:N \group_align_safe_end:
        \exp_not:n {#4}
      }
    \cs_set_eq:NN \__peek_true:w #1
    \cs_set:Npe \__peek_false:w
      {
        \exp_not:N \group_align_safe_end:
        \exp_not:n {#5}
      }
    \peek_after:Nw #2
  }
```

(*End of definition for* `\__peek_token_generic_aux:NNNTF`.)

`\__peek_token_generic:NNTF`
`\__peek_token_remove_generic:NNTF`

For token removal there needs to be a call to the auxiliary function which does the work.

```
\cs_new_protected:Npn \__peek_token_generic:NNTF
  { \__peek_token_generic_aux:NNNTF \__peek_true_aux:w }
\cs_new_protected:Npn \__peek_token_generic:NNT #1#2#3
  { \__peek_token_generic:NNTF #1 #2 {#3} { } }
\cs_new_protected:Npn \__peek_token_generic:NNF #1#2#3
  { \__peek_token_generic:NNTF #1 #2 { } {#3} }
\cs_new_protected:Npn \__peek_token_remove_generic:NNTF
  { \__peek_token_generic_aux:NNNTF \__peek_true_remove:w }
\cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
  { \__peek_token_remove_generic:NNTF #1 #2 {#3} { } }
```

```
\cs_new_protected:Npn \__peek_token_remove_generic:NNF #1#2#3
  { \__peek_token_remove_generic:NNTF #1 #2 { } {#3} }
```

(*End of definition for* `\__peek_token_generic:NNTF` *and* `\__peek_token_remove_generic:NNTF`.)

`\__peek_execute_branches_meaning:`

The meaning test is straight forward.

```
\cs_new:Npn \__peek_execute_branches_meaning:
  {
    \if_meaning:w \l_peek_token \l__peek_search_token
      \exp_after:wN \__peek_true:w
    \else:
      \exp_after:wN \__peek_false:w
    \fi:
  }
```

(*End of definition for* `\__peek_execute_branches_meaning:`.)

`\__peek_execute_branches_catcode:`
`\__peek_execute_branches_charcode:`
`\__peek_execute_branches_catcode_aux:`
`\__peek_execute_branches_catcode_auxii:N`
`\__peek_execute_branches_catcode_auxiii:`

The catcode and charcode tests are very similar, and in order to use the same auxiliaries we do something a little bit odd, firing `\if_catcode:w` and `\if_charcode:w` before finding the operands for those tests, which are only given in the `auxii:N` and `auxiii:` auxiliaries. For our purposes, three kinds of tokens may follow the peeking function:

- control sequences which are not equal to a non-active character token (*e.g.*, macro, primitive);
- active characters which are not equal to a non-active character token (*e.g.*, macro, primitive);
- explicit non-active character tokens, or control sequences or active characters set equal to a non-active character token.

The first two cases are not distinguishable simply using TEX's `\futurelet`, because we can only access the `\meaning` of tokens in that way. In those cases, detected thanks to a comparison with `\scan_stop:`, we grab the following token, and compare it explicitly with the explicit search token stored in `\l__peek_search_tl`. The `\exp_not:N` prevents outer macros (coming from non-LATEX3 code) from blowing up. In the third case, `\l_peek_token` is good enough for the test, and we compare it again with the explicit search token. Just like the peek token, the search token may be of any of the three types above, hence the need to use the explicit token that was given to the peek function.

```
\cs_new:Npn \__peek_execute_branches_catcode:
  { \if_catcode:w \__peek_execute_branches_catcode_aux: }
\cs_new:Npn \__peek_execute_branches_charcode:
  { \if_charcode:w \__peek_execute_branches_catcode_aux: }
\cs_new:Npn \__peek_execute_branches_catcode_aux:
  {
        \if_catcode:w \exp_not:N \l_peek_token \scan_stop:
          \exp_after:wN \exp_after:wN
          \exp_after:wN \__peek_execute_branches_catcode_auxii:N
          \exp_after:wN \exp_not:N
        \else:
          \exp_after:wN \__peek_execute_branches_catcode_auxiii:
        \fi:
  }
\cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1
```

```
  {
        \exp_not:N #1
        \exp_after:wN \exp_not:N \l__peek_search_tl
      \exp_after:wN \__peek_true:w
    \else:
      \exp_after:wN \__peek_false:w
    \fi:
    #1
  }
\cs_new:Npn \__peek_execute_branches_catcode_auxiii:
  {
        \exp_not:N \l_peek_token
        \exp_after:wN \exp_not:N \l__peek_search_tl
      \exp_after:wN \__peek_true:w
    \else:
      \exp_after:wN \__peek_false:w
    \fi:
  }
```

(*End of definition for* `\__peek_execute_branches_catcode:` *and others.*)

`\peek_catcode:NTF`
`\peek_catcode_remove:NTF`
`\peek_charcode:NTF`
`\peek_charcode_remove:NTF`
`\peek_meaning:NTF`
`\peek_meaning_remove:NTF`

The public functions themselves cannot be defined using `\prg_new_conditional:Npnn`. Instead, the `TF`, `T`, `F` variants are defined in terms of corresponding variants of `\__peek_token_generic:NNTF` or `\__peek_token_remove_generic:NNTF`, with first argument one of `\__peek_execute_branches_catcode:`, `\__peek_execute_branches_charcode:`, or `\__peek_execute_branches_meaning:`.

```
\tl_map_inline:nn { { catcode } { charcode } { meaning } }
  {
    \tl_map_inline:nn { { } { _remove } }
      {
        \tl_map_inline:nn { { TF } { T } { F } }
          {
            \cs_new_protected:cpe { peek_ #1 ##1 :N ####1 }
              {
                \exp_not:c { __peek_token ##1 _generic:NN ####1 }
                \exp_not:c { __peek_execute_branches_ #1 : }
              }
          }
      }
  }
```

(*End of definition for* `\peek_catcode:NTF` *and others. These functions are documented on page 205.*)

`\peek_N_type:TF`
`\__peek_execute_branches_N_type:`
`\__peek_N_type:w`
`\__peek_N_type_aux:nnw`

All tokens are `N`-type tokens, except in four cases: begin-group tokens, end-group tokens, space tokens with character code 32, and outer tokens. Since `\l_peek_token` might be outer, we cannot use the convenient `\bool_if:nTF` function, and must resort to the old trick of using `\ifodd` to expand a set of tests. The `false` branch of this test is taken if the token is one of the first three kinds of non-`N`-type tokens (explicit or implicit), thus we call `\__peek_false:w`. In the `true` branch, we must detect outer tokens, without impacting performance too much for non-outer tokens. The first filter is to search for `outer` in the `\meaning` of `\l_peek_token`. If that is absent, `\__peek_use_none_delimit_by_s_stop:w` cleans up, and we call `\__peek_true:w`. Otherwise, the token can be a non-outer macro or a primitive mark whose parameter or replacement text contains `outer`, it can be the primitive `\outer`, or it can be an outer token. Macros and marks would have ma in the part before the first occurrence of outer; the meaning of \outer has nothing after outer, contrarily to outer macros; and that covers all cases, calling __peek_-true:w or __peek_false:w as appropriate. Here, there is no ⟨*search token*⟩, so we feed a dummy \scan_stop: to the __peek_token_generic:NNTF function.

```
\group_begin:
  \cs_set_protected:Npn \__peek_tmp:w #1 \s__peek_stop
    {
      \cs_new_protected:Npn \__peek_execute_branches_N_type:
        {
          \if_int_odd:w
              \if_catcode:w \exp_not:N \l_peek_token {   \c_zero_int \fi:
              \if_catcode:w \exp_not:N \l_peek_token }   \c_zero_int \fi:
              \if_meaning:w \l_peek_token \c_space_token \c_zero_int \fi:
              \c_one_int
            \exp_after:wN \__peek_N_type:w
              \token_to_meaning:N \l_peek_token
              \s__peek_mark \__peek_N_type_aux:nnw
              #1 \s__peek_mark \__peek_use_none_delimit_by_s_stop:w
              \s__peek_stop
            \exp_after:wN \__peek_true:w
          \else:
            \exp_after:wN \__peek_false:w
          \fi:
        }
      \cs_new_protected:Npn \__peek_N_type:w ##1 #1 ##2 \s__peek_mark ##3
        { ##3 {##1} {##2} }
    }
  \exp_after:wN \__peek_tmp:w \tl_to_str:n { outer } \s__peek_stop
\group_end:
\cs_new_protected:Npn \__peek_N_type_aux:nnw #1 #2 #3 \fi:
  {
    \fi:
    \tl_if_in:noTF {#1} { \tl_to_str:n {ma} }
      { \__peek_true:w }
      { \tl_if_empty:nTF {#2} { \__peek_true:w } { \__peek_false:w } }
  }
\cs_new_protected:Npn \peek_N_type:TF
  {
    \__peek_token_generic:NNTF
      \__peek_execute_branches_N_type: \scan_stop:
  }
\cs_new_protected:Npn \peek_N_type:T
  { \__peek_token_generic:NNT \__peek_execute_branches_N_type: \scan_stop: }
\cs_new_protected:Npn \peek_N_type:F
  { \__peek_token_generic:NNF \__peek_execute_branches_N_type: \scan_stop: }
```

(*End of definition for* \peek_N_type:TF *and others. This function is documented on page* 206*.*)

```
⟨/tex⟩
⟨/package⟩
```

Chapter 63

l3prop implementation

The following test files are used for this code: m3prop001, m3prop002, m3prop003, m3prop004, m3show001*.*

```
⟨*package⟩
⟨@@=prop⟩
```

With the (default) flat data storage, a property list is a macro whose top-level expansion is of the form

> `\s__prop \__prop_chk:w \__prop_pair:wn` ⟨key_1⟩ `\s__prop {`⟨$value_1$⟩`}`
> ...
> `\__prop_pair:wn` ⟨key_n⟩ `\s__prop {`⟨$value_n$⟩`}`

where `\s__prop` is a scan mark (equal to `\scan_stop:`), `\__prop_chk:w` produces a suitable error if the property list is used directly in the input stream, and `\__prop_pair:wn` can be used to map through the property list.

With the linked data storage, each property list entry ⟨key_i⟩–⟨$value_i$⟩ is stored into a token list `\__prop` ⟨*prefix*⟩ ⟨key_i⟩. The ⟨*prefix*⟩ is one or more characters (no spaces), constructed automatically only once, when the property list is initially declared. The control sequence name does not conform to standard naming for variables because (1) this is an internal control sequence, not really a expl3 variable; (2) keeping track of the scope `l` or `g` throughout all functions would be a pretty big mess, especially if users accidentally mix local and global use (we would have to always check for such mistakes, rather than only checking when suitable debug options are set); (3) shorter control sequence names use less memory and are quicker in case of hash collisions, which may matter since we are using many control sequences.

We need to enable mapping through such a property list, but without storing a list of all entries anywhere: this is achieved by making each of these token lists also store a pointer to the next entry. To enable efficient deletion, the token lists also store a pointer to the previous entry. This means we have a doubly-linked list. To avoid having to special-case the two ends of the doubly-linked list when deleting entries, we include as a zeroth entry in the doubly-linked list the property list variable itself, and we include as an $(n + 1)$-th entry in the doubly-linked list an end-pointer `\__prop` ⟨*prefix*⟩ (no trailing space, so it differs from an empty key). The space before ⟨*prefix*⟩ ensures there is no collision with other l3prop internal functions, even if we have very many linked property lists being defined.

The property list variable itself is a token list of the form

`\__prop_flatten:w \__prop` ⟨*prefix*⟩ `\s__prop {`⟨*prefix*⟩`} \__prop` ⟨*prefix*⟩ ⟨*key*$_1$⟩

Here, `\__prop_flatten:w` serves as an efficiently recognized marker, and when `f`-expanded it is tasked with fully unpacking the property list into the same form as the default data storage so as to ease conversion. The ⟨*prefix*⟩ is used when looking up an entry. The token list `\__prop` ⟨*prefix*⟩ (see below) contains a pointer to the last key to help insert a new entry. The pointer to ⟨*key*$_1$⟩ is needed to start a mapping. The token list labeled by ⟨*key*$_i$⟩ is of the form

`\use_none:n \__prop` ⟨*prefix*⟩ ⟨*key*$_{i-1}$⟩ `\__prop_pair:wn` ⟨*key*$_i$⟩ `\s__prop {`⟨*value*$_i$⟩`} \__prop` ⟨*prefix*⟩ ⟨*key*$_{i+1}$⟩

where the pointer to ⟨*key*$_{i-1}$⟩ is needed when deleting the ⟨*key*$_i$⟩. Expanding this will run `\__prop_pair:wn` on the ⟨*key*$_i$⟩–⟨*value*$_i$⟩ pair (for speed, ⟨*key*$_i$⟩ is kept as explicit tokens rather than slowly extracting it from a control sequence name), then move on to the next key, thus mapping through the whole list. The mapping is ended upon expanding `\__prop` ⟨*prefix*⟩, which is the token list

`\use_none:n \__prop` ⟨*prefix*⟩ ⟨*key*$_n$⟩

Let us think about deleting the ⟨*key*$_i$⟩. We need to update the ⟨*key*$_{i-1}$⟩ and ⟨*key*$_{i+1}$⟩ to point to each other instead of ⟨*key*$_i$⟩. To edit the corresponding token lists, it is important that `\__prop` ⟨*prefix*⟩ ⟨*key*$_i$⟩ be at the "same place" in the token lists also in the boundary cases $i = 1$ or $i = n$, namely as the second token, or as the second argument after `\s__prop`.

63.1 Internal auxiliaries

`\__prop_tmp:w`

Scratch macro, defined as needed, for instance to save `\__prop_pair:wn` when concatenating.

```
\cs_new_eq:NN \__prop_tmp:w ?
```

(*End of definition for* `\__prop_tmp:w`.)

`\l__prop_internal_tl`

Token list used in various places: for the prefix; when converting from flat to linked props; and to store the new key–value pair inserted by `\prop_put:Nnn`.

```
\tl_new:N \l__prop_internal_tl
```

(*End of definition for* `\l__prop_internal_tl`.)

`\s__prop_mark`
`\s__prop_stop`

Internal scan marks.

```
\scan_new:N \s__prop_mark
\scan_new:N \s__prop_stop
```

(*End of definition for* `\s__prop_mark` *and* `\s__prop_stop`.)

`\q__prop_recursion_tail`
`\q__prop_recursion_stop`

Internal recursion quarks.

```
\quark_new:N \q__prop_recursion_tail
\quark_new:N \q__prop_recursion_stop
```

(*End of definition for* `\q__prop_recursion_tail` *and* `\q__prop_recursion_stop`.)

__prop_if_recursion_tail_stop:n
__prop_if_recursion_tail_stop:o

Functions to query recursion quarks.

```
\__kernel_quark_new_test:N \__prop_if_recursion_tail_stop:n
\cs_generate_variant:Nn \__prop_if_recursion_tail_stop:n { o }
```

(*End of definition for* `\__prop_if_recursion_tail_stop:n` *and* `\__prop_if_recursion_tail_stop:o`.)

63.2 Structure of a property list

\s__prop

A private scan mark is used as a marker after each key, and at the very beginning of the property list.

```
\scan_new:N \s__prop
```

(*End of definition for* `\s__prop`.)

__prop_chk:w
__prop_chk_loop:nw
__prop_chk_get:nw

This removes the flat property list from the input stream and complains about a bad use of a property list. Since property lists do not have an end-marker, we slowly peek ahead in a loop. Speed does not matter since this is for an error situation. While `\__prop_pair:wn` does not keep a fixed definition, it always includes the internal `\s__prop` in its argument specification, so that there is no risk of accidentally picking up a public token instead of `\__prop_pair:wn` when doing a meaning test. We collect the keys and values to produce a more useful error message.

```
\cs_new_protected:Npn \__prop_chk:w { \__prop_chk_loop:nw { } }
\cs_new_protected:Npn \__prop_chk_loop:nw #1
  {
    \peek_meaning:NTF \__prop_pair:wn
      { \__prop_chk_get:nw {#1} }
      { \msg_error:nne { prop } { misused } {#1} }
  }
\cs_new_protected:Npn \__prop_chk_get:nw #1 \__prop_pair:wn #2 \s__prop #3
  { \__prop_chk_loop:nw { #1 , ~ {#2} = { \tl_to_str:n {#3} } } }
```

(*End of definition for* `\__prop_chk:w`, `\__prop_chk_loop:nw`, *and* `\__prop_chk_get:nw`.)

__prop_pair:wn

Used as `\__prop_pair:wn` ⟨*key*⟩ `\s__prop` {⟨*item*⟩} for both storage types, this internal token starts each key–value pair in the property list. This default definition is changed globally by any mapping function, so there is not much point trying to make it an error. Instead, the error is produced by `\__prop_chk:w`.

```
\cs_new:Npn \__prop_pair:wn #1 \s__prop #2 { }
```

(*End of definition for* `\__prop_pair:wn`.)

__prop_flatten:w

We implement here the fact that `f`-expanding a linked property list gives a flat property list. Leaving a linked property list in the input stream will turn it into a flat property list so that the error implemented by `\__prop_chk:w` will correctly be triggered.

```
\cs_new_protected:Npn \__prop_flatten:w #1 \s__prop #2#3
  { \use:e { \__prop_flatten_aux:N #3 } }
```

(*End of definition for* `\__prop_flatten:w`.)

__prop_flatten:N
__prop_flatten_aux:w
__prop_flatten_aux:N
__prop_flatten_loop:w

The main function `\__prop_flatten:N` receives a linked property list and flattens it. The auxiliary `\__prop_flatten_aux:N` receives a pointer to the first key and flattens the linked property list into a flat property list. This is only restricted-expandable as it involves mapping through all of the property list's entries starting from ⟨*key*$_1$⟩. The looping function `\__prop_flatten_loop:w` removes `\use_none:n` and a backwards pointer `#2`, leaves the key–value pair for `\use:e` to receive, and calls itself again after expanding the next key's token list. Its argument `#3` is empty, except at the end where it is the `\use_none:nnnn` appearing in the definition of `\__prop_flatten_aux:N`, which ends the loop.

```
\cs_new:Npn \__prop_flatten:N #1
  { \exp_after:wN \__prop_flatten_aux:w #1 }
\cs_new:Npn \__prop_flatten_aux:w #1 \s__prop #2 { \__prop_flatten_aux:N }
\cs_new:Npn \__prop_flatten_aux:N #1
  {
    \s__prop \__prop_chk:w
    \exp_after:wN \__prop_flatten_loop:w #1
    \use_none:nnnn \__prop_pair:wn \s__prop { }
  }
\cs_new:Npn \__prop_flatten_loop:w #1#2#3 \__prop_pair:wn #4 \s__prop #5
  {
    #3
    \exp_not:n { \__prop_pair:wn #4 \s__prop {#5} }
    \exp_after:wN \__prop_flatten_loop:w
  }
```

(*End of definition for* `\__prop_flatten:N` *and others.*)

\g__prop_prefix_int
\c__prop_basis_int

Used to assign prefixes for each linked property list. It is converted to base `\c__prop_basis_int`, then each digit is converted to a character, starting at `!` (the character after space).

```
\int_new:N \g__prop_prefix_int
\int_const:Nn \c__prop_basis_int { \c_max_char_int - `\! }
```

(*End of definition for* `\g__prop_prefix_int` *and* `\c__prop_basis_int`*.*)

__prop_next_prefix:
__prop_to_prefix:n

Store in `\l__prop_internal_tl` the conversion of `\g__prop_prefix_int` to characters, and increment this integer for use in the next linked property list. No need to optimize since this is only used when declaring the property list the first time. The aim here is to make this string as short as we can, given the range of distinct characters available. This speeds up the work of `\cs:w` … `\cs_end:` that looks up keys in the hash table.

```
\cs_new_protected:Npn \__prop_next_prefix:
  {
    \tl_set:Ne \l__prop_internal_tl
      { \__prop_to_prefix:n { \g__prop_prefix_int } }
    \int_gincr:N \g__prop_prefix_int
  }
\cs_new:Npn \__prop_to_prefix:n #1
  {
    \int_compare:nNnTF {#1} > \c__prop_basis_int
      {
        \exp_args:Nf \__prop_to_prefix:n
          { \int_div_truncate:nn {#1} \c__prop_basis_int }
        \exp_args:Nf \__prop_to_prefix:n
```

```
            { \int_mod:nn {#1} \c__prop_basis_int }
        }
        { \char_generate:nn { `\! + #1 } { 12 } }
    }
```

(*End of definition for* `\__prop_next_prefix:` *and* `\__prop_to_prefix:n`.)

`\__prop_if_flat:NTF`
`\__prop_if_flat_aux:w`

We could either test for the presence of `\__prop_chk:w` (flat property list) or of `\__prop_flatten:w` (linked property list). We make the second choice; this way props that are accidentally `\relax` are treated as they were before. The auxiliary receives `\use_i:nn` or `\use_ii:nn` as `#3`. As a transitional fix we avoid erroring in case the prop is undefined (the `\exp_after:wN` is omitted in that case, taking the flat branch).

```
\cs_new:Npn \__prop_if_flat:NTF #1
  {
    \prop_if_exist:NT #1
    \exp_after:wN \__prop_if_flat_aux:w #1
    \s__prop_mark \use_ii:nn
    \__prop_flatten:w \s__prop_mark \use_i:nn \s__prop_stop
  }
\cs_new:Npn \__prop_if_flat_aux:w
    #1 \__prop_flatten:w #2 \s__prop_mark #3 #4 \s__prop_stop {#3}
```

(*End of definition for* `\__prop_if_flat:NTF` *and* `\__prop_if_flat_aux:w`.)

63.3 Allocation and initialisation

`\c_empty_prop`

An empty flat prop.

```
\tl_const:Nn \c_empty_prop { \s__prop \__prop_chk:w }
```

(*End of definition for* `\c_empty_prop`. *This variable is documented on page 221.*)

`\prop_new:N`
`\prop_new:c`

Flat property lists are initialized with the value `\c_empty_prop`.

```
\cs_new_protected:Npn \prop_new:N #1
  {
    \__kernel_chk_if_free_cs:N #1
    \cs_gset_eq:NN #1 \c_empty_prop
  }
\cs_generate_variant:Nn \prop_new:N { c }
```

(*End of definition for* `\prop_new:N`. *This function is documented on page 213.*)

`\prop_new_linked:N`
`\prop_new_linked:c`
`\__prop_new_linked:N`

The auxiliary is used in `\prop_make_linked:N`. For linked property lists, get a new prefix in `\l__prop_internal_tl`, then use it to set up the internal structure: the last token in `#1` is usually a pointer to the first key, which is here the end-pointer. That end-pointer has a pointer to the previous key (usually the last key), which is the variable `#1` itself that begins the doubly-linked list.

```
\cs_new_protected:Npn \prop_new_linked:N #1
  {
    \__kernel_chk_if_free_cs:N #1
    \__prop_new_linked:N #1
  }
\cs_new_protected:Npn \__prop_new_linked:N #1
  {
```

```
    \__prop_next_prefix:
    \cs_gset_nopar:Npe #1
      {
        \__prop_flatten:w
        \exp_not:c { __prop ~ \l__prop_internal_tl }
        \s__prop { \l__prop_internal_tl }
        \exp_not:c { __prop ~ \l__prop_internal_tl }
      }
    \cs_gset_nopar:cpe { __prop ~ \l__prop_internal_tl }
      {
        \exp_not:N \use_none:n
        \exp_not:N #1
      }
  }
\cs_generate_variant:Nn \prop_new_linked:N { c }
```

(*End of definition for* `\prop_new_linked:N` *and* `\__prop_new_linked:N`. *This function is documented on page 213.*)

`\prop_clear:N` `\prop_clear:c` `\prop_gclear:N` `\prop_gclear:c` `\__prop_clear:NNN` `\__prop_clear:wNNN` `\__prop_clear_loop:Nw`

Clearing a flat property list is like declaring it anew, simply setting it equal to `\c_empty_prop`. For linked property lists we must clear all of the variables storing individual keys, which requires a loop. At each step of the loop, `\__prop_clear_loop:Nw` receives `\cs_(g)set_eq:NN`, `\use_none:n`, the backwards pointer, an empty `#4` (except at the end of the loop), and the key–value pair `#5=#6` which we disregard. The looping auxiliary undefines the previous key's token list (this includes the main token list, but that is fine because it is restored at the end) and calls itself after expanding the next key's token list. The loop ends when `#4` is `\use_none:nnnn`. After the loop, `\__prop_clear:wNNN` correctly sets up the main variable `#6` and the end-pointer `#1`. Importantly, this is done using `\cs_(g)set_nopar:Npe` and `\exp_not:n` because the almost-equivalent `\tl_set:Nn` would complain in debug mode about the fact that the main variable is undefined at this stage. Importantly, `\__prop_clear_entries:NN` is used in the implementation of `\prop_set_eq:NN`.

```
\cs_new_protected:Npn \prop_clear:N
  { \__prop_clear:NNN \cs_set_eq:NN \cs_set_nopar:Npe }
\cs_generate_variant:Nn \prop_clear:N  { c }
\cs_new_protected:Npn \prop_gclear:N
  { \__prop_clear:NNN \cs_gset_eq:NN \cs_gset_nopar:Npe }
\cs_generate_variant:Nn \prop_gclear:N { c }
\cs_new_protected:Npn \__prop_clear:NNN #1#2#3
  {
    \__prop_if_flat:NTF #3
      { #1 #3 \c_empty_prop }
      { \exp_after:wN \__prop_clear:wNNN #3 #1 #2 #3 }
  }
\cs_new_protected:Npn \__prop_clear:wNNN
    \__prop_flatten:w #1 \s__prop #2#3#4#5#6
  {
    \__prop_clear_entries:NN #4 #3
    #5 #6 { \exp_not:n { \__prop_flatten:w #1 \s__prop {#2} #1 } }
    #5 #1 { \exp_not:n { \use_none:n #6 } }
  }
\cs_new_protected:Npn \__prop_clear_entries:NN #1#2
  {
```

```
    \exp_after:wN \__prop_clear_loop:Nw \exp_after:wN #1 #2
    \use_none:nnnn \__prop_pair:wn \s__prop { }
  }
\cs_new_protected:Npn \__prop_clear_loop:Nw
    #1#2#3#4 \__prop_pair:wn #5 \s__prop #6
  {
    #1 #3 \tex_undefined:D
    #4
    \exp_after:wN \__prop_clear_loop:Nw
    \exp_after:wN #1
  }
```

(*End of definition for* `\prop_clear:N` *and others. These functions are documented on page 213.*)

`\prop_clear_new:N`
`\prop_clear_new:c`
`\prop_gclear_new:N`
`\prop_gclear_new:c`
`\prop_clear_new_linked:N`
`\prop_clear_new_linked:c`
`\prop_gclear_new_linked:N`
`\prop_gclear_new_linked:c`

A simple variation of the token list functions.

```
\cs_new_protected:Npn \prop_clear_new:N  #1
  { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
\cs_generate_variant:Nn \prop_clear_new:N  { c }
\cs_new_protected:Npn \prop_gclear_new:N #1
  { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
\cs_generate_variant:Nn \prop_gclear_new:N { c }
\cs_new_protected:Npn \prop_clear_new_linked:N  #1
  { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new_linked:N #1 } }
\cs_generate_variant:Nn \prop_clear_new_linked:N  { c }
\cs_new_protected:Npn \prop_gclear_new_linked:N #1
  { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new_linked:N #1 } }
\cs_generate_variant:Nn \prop_gclear_new_linked:N { c }
```

(*End of definition for* `\prop_clear_new:N` *and others. These functions are documented on page 213.*)

`\prop_set_eq:NN`
`\prop_set_eq:cN`
`\prop_set_eq:Nc`
`\prop_set_eq:cc`
`\prop_gset_eq:NN`
`\prop_gset_eq:cN`
`\prop_gset_eq:Nc`
`\prop_gset_eq:cc`
`\__prop_set_eq:NNNN`
`\__prop_set_eq:wNNNN`
`\__prop_set_eq:nNnNN`
`\__prop_set_eq_loop:NNnw`
`\__prop_set_eq_end:w`

If both variables are accidentally the same variable (or equal flat property lists, as it turns out) we do nothing, otherwise the following code would lose all entries. If the target variable #3 is a flat prop, either copy directly or flatten before copying. If it is a linked prop, we must clear it, then go through the entries in #4 to add them to #3.

```
\cs_new_protected:Npn \prop_set_eq:NN
  { \__prop_set_eq:NNNN \cs_set_eq:NN \cs_set_nopar:Npe }
\cs_generate_variant:Nn \prop_set_eq:NN { Nc , cN , cc }
\cs_new_protected:Npn \prop_gset_eq:NN
  { \__prop_set_eq:NNNN \cs_gset_eq:NN \cs_gset_nopar:Npe }
\cs_generate_variant:Nn \prop_gset_eq:NN { Nc , cN , cc }
\cs_new_protected:Npn \__prop_set_eq:NNNN #1#2#3#4
  {
    \cs_if_eq:NNF #3#4
      {
        \__prop_if_flat:NTF #3
          {
            \__prop_if_flat:NTF #4
              { #1 #3 #4 }
              { #2 #3 { \__prop_flatten:N #4 } }
          }
          { \exp_after:wN \__prop_set_eq:wNNNN #3 #1#2#3#4 }
      }
  }
\cs_new_protected:Npn \__prop_set_eq:wNNNN
```

```
    \__prop_flatten:w #1 \s__prop #2#3#4#5#6#7
  {
    \__prop_clear_entries:NN #4 #3
    \exp_args:Nf \__prop_set_eq:nNnNN {#7} #1 {#2} #5 #6
  }
```

We have used that f-expanding either type of prop gives a flat prop. At this stage `\__prop_set_eq:nNnNN` receives the second variable as a flat prop, the end-pointer, the prefix, the suitable `\cs_(g)set_nopar:Npe` assignment, and the first variable itself. Remove the leading `\s__prop` and `\__prop_chk:w` with `\use_i:nnn`, then start the loop.

```
\cs_new_protected:Npn \__prop_set_eq:nNnNN #1#2#3#4#5
  {
    \use_i:nnn
      {
        \__prop_set_eq_loop:NNnw #5 #4 {#3}
        \__prop_flatten:w #2 \s__prop {#3}
      }
      #1
    \use_none:n \__prop_pair:wn ? \s__prop
  }
```

The looping function receives the current pointer `#1` (initially the variable itself), the defining function `#2` and the prefix `#3`, then a partial definition `#4` (which in later stages includes the backwards pointer), followed by the current value as `\s__prop {#5}`. It seeks the next key `#7` to construct in `\l__prop_internal_tl` the next pointer `\__prop` ⟨*prefix*⟩ ⟨*next key*⟩ (the argument `#6` is empty, except at the end of the loop, where it is `\use_none:n` in such a way as to delete the ⟨*space*⟩ and ⟨*next key*⟩). Then the token list (current pointer) `#1` is set-up to contain the partial definition and current value, as well as the newly constructed next pointer. After a line responsible for correctly ending the loop with `\__prop_set_eq_end:w`, we loop, setting up the next definition, which starts with `\use_none:n` and a backwards pointer to `#1` followed by the ⟨*next key*⟩ `#7` and so on.

```
\cs_new_protected:Npn \__prop_set_eq_loop:NNnw
    #1#2#3#4 \s__prop #5#6 \__prop_pair:wn #7 \s__prop
  {
    \tl_set:Ne \l__prop_internal_tl { \exp_not:c { __prop ~ #3 #6 ~ #7 } }
    #2 #1 { \exp_not:n { #4 \s__prop {#5} } \exp_not:o \l__prop_internal_tl }
    \use_none:n #6 \__prop_set_eq_end:w
    \exp_after:wN \__prop_set_eq_loop:NNnw \l__prop_internal_tl #2 {#3}
    \use_none:n #1 \__prop_pair:wn #7 \s__prop
  }
```

The end-code picks up what is needed to correctly assign the last token list (the end pointer), which is simply `\use_none:n \__prop_`⟨*prefix*⟩⟨*space*⟩⟨*key*$_n$⟩.

```
\cs_new_protected:Npn \__prop_set_eq_end:w
    \exp_after:wN \__prop_set_eq_loop:NNnw #1#2#3
    \use_none:n #4#5 \s__prop
  {
    \exp_after:wN #2 \l__prop_internal_tl { \exp_not:n { \use_none:n #4 } }
  }
```

(*End of definition for* `\prop_set_eq:NN` *and others. These functions are documented on page 213.*)

`\prop_make_flat:N` `\prop_make_flat:c` `\__prop_make_flat:Nn`

The only interesting case is when given a linked prop. Clear the linked property list using `\__prop_clear:wNNN` with local assignments (it does not matter since we are at the outermost group level, and `\cs_set_eq:NN` is very slightly faster than its global version. Then store the contents (expanded preventively by `\exp_args:NNf`) with an assignment `\cs_set_nopar:Npe` that does not perform l3debug checks.

```
\cs_new_protected:Npn \prop_make_flat:N #1
  {
    \int_compare:nNnTF { \tex_currentgrouplevel:D } = 0
      {
        \__prop_if_flat:NTF #1 { }
          { \exp_args:NNf \__prop_make_flat:Nn #1 {#1} }
      }
      {
        \msg_error:nnee { prop } { inner-make }
          { \token_to_str:N \prop_make_flat:N } { \token_to_str:N #1 }
      }
  }
\cs_generate_variant:Nn \prop_make_flat:N { c }
\cs_new_protected:Npn \__prop_make_flat:Nn #1#2
  {
    \exp_after:wN \__prop_clear:wNNN #1 \cs_set_eq:NN \cs_set_nopar:Npe #1
    \cs_set_nopar:Npe #1 { \exp_not:n {#2} }
  }
```

(*End of definition for* `\prop_make_flat:N` *and* `\prop_make_flat:c` `\__prop_make_flat:Nn`. *This function is documented on page* 214.)

`\prop_make_linked:N`
`\prop_make_linked:c`
`\__prop_make_linked:Nn`

The only interesting case is when given a flat prop. We expand the contents for later use. Then `\__prop_new_linked:N` disregards that previous value of `#1` and initializes the linked prop. We can then use an auxiliary `\__prop_set_eq:wNNNN` underlying `\prop_set_eq:NN`, with the prop contents saved as `\l__prop_internal_tl`. That step is a bit unsafe, as `\l__prop_internal_tl` (really, a flat prop here) is used within `\__prop_set_eq:wNNNN` itself, but it is in fact expanded early enough to be ok.

```
\cs_new_protected:Npn \prop_make_linked:N #1
  {
    \int_compare:nNnTF { \tex_currentgrouplevel:D } = 0
      {
        \__prop_if_flat:NTF #1
          { \exp_args:NNo \__prop_make_linked:Nn #1 {#1} } { }
      }
      {
        \msg_error:nnee { prop } { inner-make }
          { \token_to_str:N \prop_make_linked:N } { \token_to_str:N #1 }
      }
  }
\cs_generate_variant:Nn \prop_make_linked:N { c }
\cs_new_protected:Npn \__prop_make_linked:Nn #1#2
  {
    \__prop_new_linked:N #1
    \tl_set:Nn \l__prop_internal_tl {#2}
    \exp_after:wN \__prop_set_eq:wNNNN #1
      \cs_set_eq:NN \cs_set_nopar:Npe #1 \l__prop_internal_tl
  }
```

(*End of definition for* `\prop_make_linked:N` *and* `\__prop_make_linked:Nn`. *This function is documented on page* 214.)

\l_tmpa_prop
\l_tmpb_prop
\g_tmpa_prop
\g_tmpb_prop

We can now initialize the scratch variables.

```
\prop_new:N \l_tmpa_prop
\prop_new:N \l_tmpb_prop
\prop_new:N \g_tmpa_prop
\prop_new:N \g_tmpb_prop
```

(*End of definition for* `\l_tmpa_prop` *and others. These variables are documented on page 221.*)

\prop_concat:NNN
\prop_concat:ccc
\prop_gconcat:NNN
\prop_gconcat:ccc
__prop_concat:NNNNN
__prop_concat:nNNN

The basic strategy is to copy the first variable into the target, then loop through the second variable, calling `\prop_(g)put:Nnn` on each item. To avoid running the l3debug scope checks on each of these steps, we use the auxiliaries that underly `\prop_set_eq:NN` and `\prop_put:Nnn`, whose syntax is a bit unwieldy. We work directly with the target prop `#3` as a scratch space, because copying over from a temporary variable to `#3` would be slow in the linked case. If `#5` is `#3` itself we have to be careful not to lose the data, and we even take the opportunity to skip the copying step completely. To keep the correct version of the duplicate keys we use the code underlying `\prop_put_if_not_in:Nnn`, which involves passing `\use_none:nnn` to the auxiliary instead of nothing. There is no need to check for the case where `#3` is equal to `#4` because in that case `\prop_(g)set_eq:NN #3 #4` (or rather the underlying auxiliary) is correctly set up to do no needless work.

```
\cs_new_protected:Npn \prop_concat:NNN
  { \__prop_concat:NNNNN \cs_set_eq:NN \cs_set_nopar:Npe }
\cs_generate_variant:Nn \prop_concat:NNN { ccc }
\cs_new_protected:Npn \prop_gconcat:NNN
  { \__prop_concat:NNNNN \cs_gset_eq:NN \cs_gset_nopar:Npe }
\cs_generate_variant:Nn \prop_gconcat:NNN { ccc }
\cs_new_protected:Npn \__prop_concat:NNNNN #1#2#3#4#5
  {
    \cs_if_eq:NNTF #3 #5
      { \__prop_concat:nNNN \use_none:nnn #2 #3 #4 }
      {
        \__prop_set_eq:NNNN #1 #2 #3 #4
        \__prop_concat:nNNN { } #2 #3 #5
      }
  }
\cs_new_protected:Npn \__prop_concat:nNNN #1#2#3#4
  {
    \cs_gset_eq:NN \__prop_tmp:w \__prop_pair:wn
    \cs_gset_protected:Npn \__prop_pair:wn ##1 \s__prop
      { \__prop_put:nNNnn {#1} #2 #3 {##1} }
    \exp_last_unbraced:Nf \use_none:nn #4
    \cs_gset_eq:NN \__prop_pair:wn \__prop_tmp:w
  }
```

(*End of definition for* `\prop_concat:NNN` *and others. These functions are documented on page 215.*)

\prop_put_from_keyval:Nn
\prop_put_from_keyval:cn
\prop_gput_from_keyval:Nn
\prop_gput_from_keyval:cn
__prop_from_keyval:nn
__prop_from_keyval:Nnn
__prop_missing_eq:n

The core is a call to `\keyval_parse:nnn`, with an error message `\__prop_missing_eq:n` for entries without `=`, and a call to (essentially) `\prop_(g)put:Nnn` for valid key–value pairs. To avoid repeated scope checks (and errors) when l3debug is active, we instead use the auxiliary underlying `\prop_put:Nnn`. Because blank keys are valid here, in contrast to l3keys, we set and restore `\l__kernel_keyval_allow_blank_keys_bool`. The key–value argument may be quite large so we avoid reading it until it is really necessary.

```
\cs_new_protected:Npn \prop_put_from_keyval:Nn #1
```

```
  { \__prop_from_keyval:nn { \__prop_put:nNNnn { } \cs_set_nopar:Npe #1 } }
\cs_generate_variant:Nn \prop_put_from_keyval:Nn { c }
\cs_new_protected:Npn \prop_gput_from_keyval:Nn #1
  { \__prop_from_keyval:nn { \__prop_put:nNNnn { } \cs_gset_nopar:Npe #1 } }
\cs_generate_variant:Nn \prop_gput_from_keyval:Nn { c }
\cs_new_protected:Npn \__prop_from_keyval:nn
  {
    \bool_if:NTF \l__kernel_keyval_allow_blank_keys_bool
      { \__prop_from_keyval:Nnn \c_true_bool }
      { \__prop_from_keyval:Nnn \c_false_bool }
  }
\cs_new_protected:Npn \__prop_from_keyval:Nnn #1#2#3
  {
    \bool_set_eq:NN \l__kernel_keyval_allow_blank_keys_bool \c_true_bool
    \keyval_parse:nnn \__prop_missing_eq:n {#2} {#3}
    \bool_set_eq:NN \l__kernel_keyval_allow_blank_keys_bool #1
  }
\cs_new_protected:Npn \__prop_missing_eq:n
  { \msg_error:nnn { prop } { prop-keyval } }
```

(*End of definition for* `\prop_put_from_keyval:Nn` *and others. These functions are documented on page 216.*)

`\prop_set_from_keyval:Nn`
`\prop_set_from_keyval:cn`
`\prop_gset_from_keyval:Nn`
`\prop_gset_from_keyval:cn`

Just empty the prop (with the auxiliary underlying `\prop_clear:N` to avoid l3debug problems) and push key–value entries using `\prop_(g)put_from_keyval:Nn`.

```
\cs_new_protected:Npn \prop_set_from_keyval:Nn #1
  {
    \__prop_clear:NNN \cs_set_eq:NN \cs_set_nopar:Npe #1
    \prop_put_from_keyval:Nn #1
  }
\cs_generate_variant:Nn \prop_set_from_keyval:Nn { c }
\cs_new_protected:Npn \prop_gset_from_keyval:Nn #1
  {
    \__prop_clear:NNN \cs_gset_eq:NN \cs_gset_nopar:Npe #1
    \prop_gput_from_keyval:Nn #1
  }
\cs_generate_variant:Nn \prop_gset_from_keyval:Nn { c }
```

(*End of definition for* `\prop_set_from_keyval:Nn` *and* `\prop_gset_from_keyval:Nn`*. These functions are documented on page 214.*)

`\prop_const_from_keyval:Nn`
`\prop_const_from_keyval:cn`
`\prop_const_linked_from_keyval:Nn`
`\prop_const_linked_from_keyval:cn`

For both flat and linked constant props, we create `#1` then use the same auxiliary as for `\prop_gput_from_keyval:Nn`. It is most natural to use the already packaged `\prop_gput:Nnn`, but that would mean doing an assignment on a supposedly constant property list. To avoid errors when l3debug is activated, we use the auxiliary underlying `\prop_gput:Nnn`.

```
\cs_new_protected:Npn \prop_const_from_keyval:Nn #1
  {
    \prop_new:N #1
    \__prop_from_keyval:nn { \__prop_put:nNNnn { } \cs_gset_nopar:Npe #1 }
  }
\cs_generate_variant:Nn \prop_const_from_keyval:Nn { c }
\cs_new_protected:Npn \prop_const_linked_from_keyval:Nn #1
  {
```

```
    \prop_new_linked:N #1
    \__prop_from_keyval:nn { \__prop_put:nNNnn { } \cs_gset_nopar:Npe #1 }
  }
\cs_generate_variant:Nn \prop_const_linked_from_keyval:Nn { c }
```

(*End of definition for* `\prop_const_from_keyval:Nn` *and* `\prop_const_linked_from_keyval:Nn`*. These functions are documented on page* *214**.*)

63.4 Accessing data in property lists

Accessing/deleting/adding entries is mostly done by `\__prop_split:NnTFn`, which must be fast because it is used in many l3prop functions. Its syntax is as follows.

> `\__prop_split:NnTFn` ⟨*property list*⟩ `{`⟨*key*⟩`}`
> `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}` `{`⟨*link code*⟩`}`

If the ⟨*property list*⟩ uses the linked data storage, then it runs the ⟨*link code*⟩, otherwise it does as follows.

It splits the ⟨*property list*⟩ at the ⟨*key*⟩, giving three token lists: the ⟨*entries before*⟩ the ⟨*key*⟩, the ⟨*value*⟩ associated with the ⟨*key*⟩ and the ⟨*entries after*⟩ the ⟨*key*⟩. Both the ⟨*entries before*⟩ and the ⟨*entries after*⟩ can be empty or consist of some number of consecutive entries `\__prop_pair:wn` ⟨key_i⟩ `\s__prop` `{`⟨$value_i$⟩`}`. If the ⟨*key*⟩ is present in the ⟨*property list*⟩ then the ⟨*true code*⟩ is left in the input stream, with `#1`, `#2`, and `#3` replaced by the ⟨*entries before*⟩, ⟨*value*⟩, and ⟨*entries after*⟩. If the ⟨*key*⟩ is not present in the ⟨*property list*⟩ then the ⟨*false code*⟩ is left in the input stream. Only the ⟨*true code*⟩ is used in the replacement text of a macro defined internally, which requires `##` doubling.

`\__prop_split:NnTFn`
`\__prop_split_aux:nNTFn`
`\__prop_split_test:wn`
`\__prop_split_flat:w`
`\__prop_split_linked:w`
`\__prop_split_wrong:Nw`

The aim is to distinguish four cases: a flat prop that contains the given ⟨*key*⟩, a flat prop that does not contain it, a linked prop, and an invalid prop. The last case includes those that are set to `\relax` by `c`-expansion, as well as unrelated token list variables since these unfortunately used to "work" in earlier implementations. In the first three cases we run the `T`, `F`, and `n` arguments, and in the last case we raise an error, set the variable to a known state (empty prop), and run the `F` code (some conditionals such as `\prop_pop:NnNTF` otherwise blow up pretty badly).

The first distinction between these cases is done by `\__prop_split_test:wn`, which looks for the argument after `\s__prop`. For a flat prop it will be `\__prop_chk:w`, which leads to running `\__prop_split_flat:w`, explained below. For a linked prop it is the prefix, consisting of characters, so we end up running `\__prop_split_linked:w`, which cleans up and selects the aforementioned `n` argument. For invalid props, or rather, variables that do not contain `\s__prop`, the argument includes `\fi:`, and we end up calling `\__prop_split_wrong:Nw`, which calls `\prop_show:N` to raise a detailed error stating how the variable is wrong.

Let us return to `\__prop_split_flat:w`. This function is defined dynamically as

> `\cs_set:Npn \__prop_split_flat:w \__prop_split_linked:w #1`
> `\__prop_pair:wn` ⟨*key*⟩ `\s__prop #2`
> `#3 \s__prop_mark #4 #5 \s__prop_stop`
> `{ #4 {`⟨*true code*⟩`} }`

Its job is to seek the ⟨*key*⟩ in the property list (known to be flat at this stage) by using an argument #1 delimited essentially by that key. If indeed the variable contained the ⟨*key*⟩, then #1 is the ⟨*extract*$_1$⟩ before the key–value pair, #2 is the ⟨*value*⟩ associated with the ⟨*key*⟩, #3 is the ⟨*extract*$_2$⟩ after the key–value pair, #4 is \use_i:nnn, and we run \use_i:nnn {⟨*true code*⟩} {⟨*false code*⟩} {⟨*link code*⟩}, selecting the ⟨*true code*⟩. Otherwise, the whole property list together with \s__prop_mark \use_i:nnn is taken in as #1, then #2 is some tokens ? \fi: __prop_split_wrong:Nw ⟨*variable*⟩ that were only useful in the case of invalid props, #3 is empty, and most importantly #4 is \use_ii:nnn. This command selects the ⟨*false code*⟩.

Note that we define __prop_split_flat:w in all cases even though it is only used in the flat case. Indeed, to avoid taking in the whole property list (which may be large) as an argument more than strictly necessary, we would have to keep the ⟨*true code*⟩ positioned before the expansion of the prop variable in order to use it in the definition. The only way to do that is to store it using an assignment so we might as well just perform the assignment that we can actually use in the flat case.

```
\cs_new_protected:Npn \__prop_split:NnTFn #1#2
  {
    \exp_after:wN \__prop_split_aux:nNTFn
    \exp_after:wN { \tl_to_str:n {#2} } #1
  }
\cs_new_protected:Npn \__prop_split_aux:nNTFn #1#2#3
  {
    \cs_set:Npn \__prop_split_flat:w \__prop_split_linked:w ##1
      \__prop_pair:wn #1 \s__prop ##2 ##3 \s__prop_mark ##4 ##5 \s__prop_stop
      { ##4 {#3} }
    \exp_after:wN \__prop_split_test:wn #2 \s__prop_mark \use_i:nnn
      \__prop_pair:wn #1 \s__prop { ? \fi: \__prop_split_wrong:Nw #2 }
      \s__prop_mark \use_ii:nnn
      \s__prop_stop
  }
\cs_new:Npn \__prop_split_flat:w { }
\cs_new_protected:Npn \__prop_split_test:wn #1 \s__prop #2
  {
    \if_meaning:w \__prop_chk:w #2 \exp_after:wN \__prop_split_flat:w \fi:
    \__prop_split_linked:w
  }
\cs_new_protected:Npn \__prop_split_linked:w #1 \s__prop_stop #2#3 {#3}
\cs_new_protected:Npn \__prop_split_wrong:Nw #1#2 \s__prop_stop #3#4
  {
    \prop_show:N #1
    \cs_gset_eq:NN #1 \c_empty_prop
    #3
  }
```

(*End of definition for* __prop_split:NnTFn *and others.*)

\prop_get:NnN
\prop_get:NVN
\prop_get:NvN
\prop_get:NeN
\prop_get:NoN
\prop_get:NxN
\prop_get:cnN
\prop_get:cVN
\prop_get:cvN
\prop_get:ceN
\prop_get:coN
\prop_get:cxN
\prop_get:cnc
\prop_get:NnN*TF*
\prop_get:NVN*TF*
\prop_get:NvN*TF*
\prop_get:NeN*TF*

Here we implement both \prop_get:NnN and its branching version through __prop_get:NnnTF. It receives the prop and key, followed by an assignment used when the value is found, ⟨*true code*⟩ to run after the assignment, and some fall-back ⟨*false code*⟩ for absent values. It relies on __prop_split:NnTFn. For a flat prop, the first four arguments of __prop_split:NnTFn are used, and run either the assignment #3{##3} and ⟨*true code*⟩ #4, or the ⟨*false code*⟩ #5.

```
\cs_new_protected:Npn \prop_get:NnN #1#2#3
  {
    \__prop_get:NnnTF #1 {#2}
      { \tl_set:Nn #3 } { } { \tl_set:Nn #3 { \q_no_value } }
  }
\cs_generate_variant:Nn \prop_get:NnN { NV , Nv , Ne , c , cV , cv , ce }
\cs_generate_variant:Nn \prop_get:NnN { No , Nx , co , cx }
\cs_generate_variant:Nn \prop_get:NnN { cnc }
\prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
  {
    \__prop_get:NnnTF #1 {#2}
      { \tl_set:Nn #3 } \prg_return_true: \prg_return_false:
  }
\prg_generate_conditional_variant:Nnn \prop_get:NnN
  { NV , Nv , Ne , c , cV , cv , ce } { T , F , TF }
\prg_generate_conditional_variant:Nnn \prop_get:NnN
  { No , Nx , co , cx } { T , F , TF }
\prg_generate_conditional_variant:Nnn \prop_get:NnN
  { cnc } { T , F , TF }
\cs_new_protected:Npn \__prop_get:NnnTF #1#2#3#4#5
  {
    \__prop_split:NnTFn #1 {#2}
      { #3 {##2} #4 }
      {#5}
      { \exp_after:wN \__prop_get_linked:w #1 {#2} {#3} {#4} {#5} }
  }
```

For a linked prop we must work a bit: `\__prop_get_linked:w` is followed by the expansion of the prop, then by four brace groups: the key `#4`, the assignment code `#5`, ⟨*true code*⟩ `#6`, and ⟨*false code*⟩ `#7`. If the key is present, its value is stored in the token list `\__prop_#2~#4`. If that token list exists, `\__prop_get_linked_aux:w` gets called followed by the expansion of that token list and we grab as `#2` the value associated to that key, which we feed to the assignment code and follow-up code. If the key is absent the token list can be `\undefined` or `\relax`. In both cases `\__prop_get_linked_aux:w` finds an empty brace group as `#2`, `\use_none:n` as `#4` and the ⟨*false code*⟩ as `#5`. Note that we made `\__prop_get_linked:w` and subsequent auxiliaries expandable, because they are also used in `\prop_item:Nn`.

```
\cs_new:Npn \__prop_get_linked:w
    \__prop_flatten:w #1 \s__prop #2#3#4#5#6#7
  {
    \if_cs_exist:w __prop ~ #2 ~ \tl_to_str:n {#4} \cs_end:
      \exp_after:wN \exp_after:wN \exp_after:wN \__prop_get_linked_aux:w
      \cs:w __prop ~ #2 ~ \tl_to_str:n {#4} \exp_after:wN \cs_end:
    \else:
      \exp_after:wN \__prop_get_linked_aux:w
    \fi:
    \s__prop_mark {#5} {#6}
    \s__prop { } \s__prop_mark \use_none:n {#7}
    \s__prop_stop
  }
\cs_new:Npn \__prop_get_linked_aux:w
    #1 \s__prop #2 #3 \s__prop_mark #4 #5 #6 \s__prop_stop { #4 {#2} #5 }
```

(*End of definition for* `\prop_get:NnN` *and others. These functions are documented on page 216.*)

\prop_item:Nn
\prop_item:NV
\prop_item:No
\prop_item:Ne
\prop_item:cn
\prop_item:cV
\prop_item:co
\prop_item:ce
__prop_item:nnn

Getting the value corresponding to a key in a flat property list in an expandable fashion simply uses `\prop_map_tokens:Nn` to go through the property list. The auxiliary `\__prop_item:nnn` receives the search string `#1`, the key `#2` and the value `#3` and returns as appropriate.

```
\cs_new:Npn \prop_item:Nn #1#2
  {
    \__prop_if_flat:NTF #1
      {
        \exp_args:NNo \prop_map_tokens:Nn #1
          {
            \exp_after:wN \__prop_item:nnn
            \exp_after:wN { \tl_to_str:n {#2} }
          }
      }
      { \exp_after:wN \__prop_get_linked:w #1 {#2} \use:n { } { } }
  }
\cs_new:Npn \__prop_item:nnn #1#2#3
  {
    \str_if_eq:eeT {#1} {#2}
      { \prop_map_break:n { \exp_not:n {#3} } }
  }
\cs_generate_variant:Nn \prop_item:Nn { NV , No , Ne , c , cV , co , ce }
```

(*End of definition for* `\prop_item:Nn` *and* `\__prop_item:nnn`*. This function is documented on page 217.*)

63.5 Removing data from property lists

__prop_pop:NnNNnTF
__prop_pop_linked:wnNNnTF
__prop_pop_linked:NNNn
__prop_pop_linked:w
__prop_pop_linked_prev:w
__prop_pop_linked_next:w

This auxiliary is used by both the `\prop_pop` family and the `\prop_remove` family of functions. It receives a ⟨*prop*⟩ and a {⟨*key*⟩}, three assignment functions (`\tl_set:Nn` `\cs_set_eq:NN` `\cs_set_nopar:Npe` or their global versions), then {⟨*code*⟩} {⟨*true code*⟩} {⟨*false code*⟩}.

For a flat prop, split it. If the ⟨*key*⟩ is there, reconstruct the rest of the prop from the two extracts `##2` `##4` and assign using `\tl_(g)set:Nn`, then run ⟨*code*⟩ {⟨*value*⟩} with the ⟨*value*⟩ found, and run the ⟨*true code*⟩. If the ⟨*key*⟩ is absent, run the ⟨*false code*⟩.

For a linked prop, the removal is done by `\__prop_pop_linked:wnNNnTF`, which removes the key–value pair from the doubly-linked list and runs its last three arguments {⟨*code*⟩} {⟨*true code*⟩} {⟨*false code*⟩} depending on whether the key–value is found, in the same way as for flat props.

```
\cs_new_protected:Npn \__prop_pop:NnNNnTF #1#2#3#4#5#6#7
  {
    \__prop_split:NnTFn #1 {#2}
      {
        #4 #1 { \exp_not:n { \s__prop \__prop_chk:w ##1 ##3 } }
        #5 {##2}
        #6
      }
      {#7}
      {
        \exp_after:wN \__prop_pop_linked:wnNNnTF #1 {#2}
          #3 #4 {#5} {#6} {#7}
      }
```

```
  }
```

The next auxiliary `\__prop_pop_linked:wnNNnTF`, together with the `NNNn` auxiliary, checks if the key is present in the ⟨*linked prop*⟩, then the corresponding value (if present) is passed as a braced argument to the ⟨*code*⟩ and the ⟨*true code*⟩ or ⟨*false code*⟩ is run as appropriate. Before that, there are also three assignments: the token lists for the previous key and next key are made to point to each other, cf. `\__prop_pop_linked:w`, and the token list for the given key is made undefined.

```
\cs_new_protected:Npn \__prop_pop_linked:wnNNnTF
    \__prop_flatten:w #1 \s__prop #2#3#4#5#6#7
  {
    \if_cs_exist:w __prop ~ #2 ~ \tl_to_str:n {#4} \cs_end:
      \exp_after:wN \__prop_pop_linked:NNNn
      \cs:w __prop ~ #2 ~ \tl_to_str:n {#4} \cs_end:
      #5 #6 {#7}
    \else:
      \exp_after:wN \use_iii:nnn
    \fi:
    \use_i:nn
  }
\cs_new_protected:Npn \__prop_pop_linked:NNNn #1#2#3#4
  {
    \if_meaning:w \scan_stop: #1
      \exp_after:wN \exp_after:wN \exp_after:wN \use_iii:nnn
    \else:
      \exp_after:wN \__prop_pop_linked:w #1 #1 #2 #3 {#4}
    \fi:
  }
\cs_new_protected:Npn \__prop_pop_linked:w
    \use_none:n #1#2 \s__prop #3#4#5#6#7#8
  {
    #6 #5 \tex_undefined:D
    #7 #1
      {
        \exp_after:wN \__prop_pop_linked_prev:w #1
        \exp_not:N #4
      }
    #7 #4
      {
        \exp_not:n { \use_none:n #1 }
        \exp_not:f { \exp_after:wN \__prop_pop_linked_next:w #4 }
      }
    #8 {#3}
  }
\cs_new:Npn \__prop_pop_linked_prev:w #1 \s__prop #2#3
  { \exp_not:n { #1 \s__prop {#2} } }
\cs_new:Npn \__prop_pop_linked_next:w \use_none:n #1 { \exp_stop_f: }
```

(*End of definition for* `\__prop_pop:NnNNnTF` *and others.*)

`\prop_remove:Nn`
`\prop_remove:NV`
`\prop_remove:Ne`
`\prop_remove:cn`
`\prop_remove:cV`
`\prop_remove:ce`
`\prop_gremove:Nn`
`\prop_gremove:NV`
`\prop_gremove:Ne`
`\prop_gremove:cn`
`\prop_gremove:cV`
`\prop_gremove:ce`

Deleting from a property relies on `\__prop_pop:NnNNnTF`. The three assignment functions are suitably local or global. The last three arguments are `\use_none:n` and two empty brace groups: if the key is found we get `\use_none:n {`⟨*key*⟩`}` ⟨*empty*⟩, which expands to nothing, and otherwise we just get ⟨*empty*⟩. The auxiliary takes care of actually removing the entry from the prop.

```
\cs_new_protected:Npn \prop_remove:Nn #1#2
  {
    \__prop_pop:NnNNnTF #1 {#2}
      \cs_set_eq:NN \cs_set_nopar:Npe
      \use_none:n { } { }
  }
\cs_new_protected:Npn \prop_gremove:Nn #1#2
  {
    \__prop_pop:NnNNnTF #1 {#2}
      \cs_gset_eq:NN \cs_gset_nopar:Npe
      \use_none:n { } { }
  }
\cs_generate_variant:Nn \prop_remove:Nn  { NV , Ne , c , cV , ce }
\cs_generate_variant:Nn \prop_gremove:Nn { NV , Ne , c , cV , ce }
```

(*End of definition for* `\prop_remove:Nn` *and* `\prop_gremove:Nn`*. These functions are documented on page 217.*)

`\prop_pop:NnN`
`\prop_pop:NVN`
`\prop_pop:NoN`
`\prop_pop:cnN`
`\prop_pop:cVN`
`\prop_pop:coN`
`\prop_gpop:NnN`
`\prop_gpop:NVN`
`\prop_gpop:NoN`
`\prop_gpop:cnN`
`\prop_gpop:cVN`
`\prop_gpop:coN`
`\prop_pop:NnNTF`
`\prop_pop:NVNTF`
`\prop_pop:NoNTF`
`\prop_pop:cnNTF`
`\prop_pop:cVNTF`
`\prop_pop:coNTF`
`\prop_gpop:NnNTF`
`\prop_gpop:NVNTF`
`\prop_gpop:NoNTF`
`\prop_gpop:cnNTF`
`\prop_gpop:cVNTF`
`\prop_gpop:coNTF`

Popping a value is almost the same, but the value found is kept. For the non-branching version, we additionally set the target token list to `\q_no_value`, while for the branching version we must produce `\prg_return_true:` or `\prg_return_false:`.

```
\cs_new_protected:Npn \prop_pop:NnN #1#2#3
  {
    \__prop_pop:NnNNnTF #1 {#2}
      \cs_set_eq:NN \cs_set_nopar:Npe
      { \tl_set:Nn #3 } { } { \tl_set:Nn #3 { \q_no_value } }
  }
\cs_new_protected:Npn \prop_gpop:NnN #1#2#3
  {
    \__prop_pop:NnNNnTF #1 {#2}
      \cs_gset_eq:NN \cs_gset_nopar:Npe
      { \tl_set:Nn #3 } { } { \tl_set:Nn #3 { \q_no_value } }
  }
\cs_generate_variant:Nn \prop_pop:NnN  {     NV , No }
\cs_generate_variant:Nn \prop_pop:NnN  { c , cV , co }
\cs_generate_variant:Nn \prop_gpop:NnN {     NV , No }
\cs_generate_variant:Nn \prop_gpop:NnN { c , cV , co }
\prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
  {
    \__prop_pop:NnNNnTF #1 {#2}
      \cs_set_eq:NN \cs_set_nopar:Npe
      { \tl_set:Nn #3 } \prg_return_true: \prg_return_false:
  }
\prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
  {
    \__prop_pop:NnNNnTF #1 {#2}
      \cs_gset_eq:NN \cs_gset_nopar:Npe
      { \tl_set:Nn #3 } \prg_return_true: \prg_return_false:
  }
\prg_generate_conditional_variant:Nnn \prop_pop:NnN
  { NV , No , c , cV , co } { T , F , TF }
\prg_generate_conditional_variant:Nnn \prop_gpop:NnN
```

```
  { NV , No , c , cV , co } { T , F , TF }
```

(*End of definition for* `\prop_pop:NnN` *and others. These functions are documented on page* 216.)

63.6 Adding data to property lists

`\prop_put:Nnn`
`\prop_put:NnV`
`\prop_put:Nnv`
`\prop_put:Nne`
`\prop_put:NVn`
`\prop_put:NVV`
`\prop_put:NVv`
`\prop_put:NVe`
`\prop_put:Nvn`
`\prop_put:NvV`
`\prop_put:Nvv`
`\prop_put:Nve`
`\prop_put:Nen`
`\prop_put:NeV`
`\prop_put:Nev`
`\prop_put:Nee`
`\prop_put:Nno`
`\prop_put:Non`
`\prop_put:Noo`
`\prop_put:Nnx`
`\prop_put:NVx`
`\prop_put:NxV`
`\prop_put:Nxx`
`\prop_put:cnn`
`\prop_put:cnV`
`\prop_put:cnv`
`\prop_put:cne`
`\prop_put:cVn`
`\prop_put:cVV`
`\prop_put:cVv`
`\prop_put:cVe`
`\prop_put:cvn`
`\prop_put:cvV`
`\prop_put:cvv`
`\prop_put:cve`
`\prop_put:cen`
`\prop_put:ceV`
`\prop_put:cev`
`\prop_put:cee`
`\prop_put:cno`
`\prop_put:con`
`\prop_put:coo`
`\prop_put:cnx`
`\prop_put:cVx`
`\prop_put:cxV`
`\prop_put:cxx`
`\prop_gput:Nnn`
`\prop_gput:NnV`
`\prop_gput:Nnv`
`\prop_gput:Nne`
`\prop_gput:NVn`
`\prop_gput:NVV`
`\prop_gput:NVv`
`\prop_gput:NVe`

All of the `\prop_(g)put(_if_new):Nnn` functions are based on the same auxiliary, which receives ⟨*code*⟩ and an "assignment", followed by ⟨*prop*⟩ {⟨*key*⟩} {⟨*new value*⟩}. The assignment `\cs_(g)set_nopar:Npe` is the primitive assignment without any checking: in the case of linked props it is applied to individual pieces of the linked prop, which are typically not yet defined. Debugging the scope of the variable is done at a higher level by letting l3debug change `\prop_put:Nnn` and friends. This allows other l3prop commands to directly call the underlying auxiliary to skip this checking step and avoid getting multiple error messages for the same error. The ⟨*code*⟩ (empty for `put` and `\use_none:nnn` for `put_if_not_in`) is placed before the assignment in cases where the key is already present, in order to suppress the assignment in the `put_if_not_in` case.

```
\cs_new_protected:Npn \prop_put:Nnn
  { \__prop_put:nNNnn { } \cs_set_nopar:Npe }
\cs_new_protected:Npn \prop_gput:Nnn
  { \__prop_put:nNNnn { } \cs_gset_nopar:Npe }
\cs_new_protected:Npn \prop_put_if_not_in:Nnn
  { \__prop_put:nNNnn \use_none:nnn \cs_set_nopar:Npe }
\cs_new_protected:Npn \prop_gput_if_not_in:Nnn
  { \__prop_put:nNNnn \use_none:nnn \cs_gset_nopar:Npe }
\cs_generate_variant:Nn \prop_put:Nnn
  {
          NnV , Nnv , Nne , NV , NVV , NVv , NVe ,
    Nv , NvV , Nvv , Nve , Ne , NeV , Nev , Nee
  }
\cs_generate_variant:Nn \prop_put:Nnn
  { Nno , No , Noo , Nnx , NVx , NxV , Nxx }
\cs_generate_variant:Nn \prop_put:Nnn
  {
    c   , cnV , cnv , cne , cV , cVV , cVv , cVe ,
    cv , cvV , cvv , cve , ce , ceV , cev , cee
  }
\cs_generate_variant:Nn \prop_put:Nnn
  { cno , co , coo , cnx , cVx , cxV , cxx }
\cs_generate_variant:Nn \prop_gput:Nnn
  {
          NnV , Nnv , Nne , NV , NVV , NVv , NVe ,
    Nv , NvV , Nvv , Nve , Ne , NeV , Nev , Nee
  }
\cs_generate_variant:Nn \prop_gput:Nnn
  { Nno , No , Noo , Nnx , NVx , NxV , Nxx }
\cs_generate_variant:Nn \prop_gput:Nnn
  {
    c   , cnV , cnv , cne , cV , cVV , cVv , cVe ,
    cv , cvV , cvv , cve , ce , ceV , cev , cee
  }
\cs_generate_variant:Nn \prop_gput:Nnn
  { cno , co , coo , cnx , cVx , cxV , cxx }
```

```
\cs_generate_variant:Nn \prop_put_if_not_in:Nnn
  {
         NnV , Nnv , Nne , NV , NVV , NVv , NVe ,
    Nv , NvV , Nvv , Nve , Ne , NeV , Nev , Nee ,
    c  , cnV , cnv , cne , cV , cVV , cVv , cVe ,
    cv , cvV , cvv , cve , ce , ceV , cev , cee
  }
\cs_generate_variant:Nn \prop_gput_if_not_in:Nnn
  {
         NnV , Nnv , Nne , NV , NVV , NVv , NVe ,
    Nv , NvV , Nvv , Nve , Ne , NeV , Nev , Nee ,
    c  , cnV , cnv , cne , cV , cVV , cVv , cVe ,
    cv , cvV , cvv , cve , ce , ceV , cev , cee
  }
```

Since the true branch of `\__prop_split:NnTFn` is used as the replacement text of an internal macro, and since the ⟨*key*⟩ and new ⟨*value*⟩ may contain arbitrary tokens, it is not safe to include them in the argument of `\__prop_split:NnTFn`. We thus start by storing in `\l__prop_internal_tl` tokens which (after x-expansion) encode the key–value pair. This variable can safely be used in `\__prop_split:NnTFn`. For a flat prop, if the ⟨*key*⟩ was absent, append the new key–value to the list; otherwise concatenate the extracts `##2` and `##4` with the new key–value pair `\l__prop_internal_tl`. The updated entry is placed at the same spot as the original ⟨*key*⟩ in the property list, preserving the order of entries. For a linked prop, call `\__prop_put_linked:wnNN`, which constructs the control sequence in which we will place the new value. If it matches `\scan_stop:` then the key was not yet there and we add it using `\__prop_put_linked_new:w`, otherwise it was already there and we use `\__prop_put_linked_old:w`.

```
\cs_new_protected:Npn \__prop_put:nNNnn #1#2#3#4#5
  {
    \tl_set:Nn \l__prop_internal_tl
      {
        \exp_not:N \__prop_pair:wn \tl_to_str:n {#4}
        \s__prop { \exp_not:n {#5} }
      }
    \__prop_split:NnTFn #3 {#4}
      {
        #1 #2 #3
          {
            \s__prop \__prop_chk:w \exp_not:n {##1}
            \l__prop_internal_tl \exp_not:n {##3}
          }
      }
      { #2 #3 { \exp_not:o {#3} \l__prop_internal_tl } }
      { \exp_after:wN \__prop_put_linked:wnnN #3 {#4} {#1} #2 }
  }
\cs_new_protected:Npn \__prop_put_linked:wnnN
    \__prop_flatten:w #1 \s__prop #2#3#4
  {
    \exp_after:wN \__prop_put_linked:NNnN
    \cs:w __prop ~ #2 ~ \tl_to_str:n {#4} \cs_end:
    #1
  }
\cs_new_protected:Npn \__prop_put_linked:NNnN #1#2#3#4
  {
```

```
    \if_meaning:w \scan_stop: #1
      \exp_after:wN \__prop_put_linked_new:w #2 #1 #2 #4
    \else:
      \exp_after:wN \__prop_put_linked_old:w #1 { #3 #4 #1 }
    \fi:
  }
```

To add a new entry, `\__prop_put_linked_new:w` receives the expansion of the end-pointer, namely `\use_none:n` ⟨*last key pointer*⟩, followed by the new key pointer `#2`, the end pointer `#3`, and an assignment function `#4`. Set up the doubly-linked list in the order `#1`, `#2`, `#3`, placing the key–value pair `\l__prop_internal_tl` in `#2`. To replace an old entry, `\__prop_put_linked_old:w` receives the expansion of that entry, and it reassigns it (`#5`) using the assignment `#6`, by simply replacing the payload `#2 \s__prop #3` by `\l__prop_internal_tl`.

```
\cs_new_protected:Npn \__prop_put_linked_new:w
    \use_none:n #1#2#3#4
  {
    #4 #1
      {
        \exp_after:wN \__prop_pop_linked_prev:w #1
        \exp_not:N #2
      }
    #4 #2
      {
        \exp_not:n { \use_none:n #1 }
        \l__prop_internal_tl
        \exp_not:N #3
      }
    #4 #3 { \exp_not:n { \use_none:n #2 } }
  }
\cs_new_protected:Npn \__prop_put_linked_old:w
    \use_none:n #1#2 \s__prop #3#4#5
  {
    #5
      {
        \exp_not:n { \use_none:n #1 }
        \l__prop_internal_tl
        \exp_not:N #4
      }
  }
```

(*End of definition for* `\prop_put:Nnn` *and others. These functions are documented on page 215.*)

63.7 Property list conditionals

`\prop_if_exist_p:N`
`\prop_if_exist_p:c`
`\prop_if_exist:NTF`
`\prop_if_exist:cTF`

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \prop_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \prop_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End of definition for* `\prop_if_exist:NTF`. *This function is documented on page 217.*)

\prop_if_empty_p:N
\prop_if_empty_p:c
\prop_if_empty:NTF
\prop_if_empty:cTF
__prop_if_empty_return:w

A flat property list is empty if it matches `\c_empty_prop`. A linked property list is empty if its second token (the end pointer) and last token (the first key pointer) are equal. There cannot be false positives because the end pointer takes the form `\use_none:n` ⟨*pointer*⟩ while the other pointers have more elaborate structure. The subtle code branch here is when a non-empty flat property list is given: then `\__prop_if_empty:w` reads the whole property list as `#1` and `#2`, `#3`, `#4` are `2`, `3`, `4`, respectively.

```
\prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \c_empty_prop
      \prg_return_true:
    \else:
      \exp_after:wN \__prop_if_empty_return:w #1
        \__prop_flatten:w 2 \s__prop 34 \s__prop_stop
    \fi:
  }
\cs_new:Npn \__prop_if_empty_return:w
    #1 \__prop_flatten:w #2 \s__prop #3#4#5 \s__prop_stop
  {
    \if_meaning:w #2 #4
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_generate_conditional_variant:Nnn \prop_if_empty:N
  { c } { p , T , F , TF }
```

(*End of definition for* `\prop_if_empty:NTF` *and* `\__prop_if_empty_return:w`. *This function is documented on page 218.*)

\prop_if_in_p:Nn
\prop_if_in_p:NV
\prop_if_in_p:Ne
\prop_if_in_p:No
\prop_if_in_p:cn
\prop_if_in_p:cV
\prop_if_in_p:ce
\prop_if_in_p:co
\prop_if_in:NnTF
\prop_if_in:NVTF
\prop_if_in:NeTF
\prop_if_in:NoTF
\prop_if_in:cnTF
\prop_if_in:cVTF
\prop_if_in:ceTF
\prop_if_in:coTF
__prop_if_in_flat:nnn

For a linked prop, use `\__prop_get_linked:w` to look up whether the control sequence constructed from the prefix and the sought-after key exists; this auxiliary calls `\use_none:n` {⟨*value*⟩} `\prg_return_true:` if the key is found, and otherwise `\prg_return_false:`. For a flat prop, testing expandably if a key is there requires to go through the key–value pairs one by one. This is rather slow, and a faster test would be

```
\prg_new_protected_conditional:Npnn \prop_if_in:Nn #1 #2
  {
    \@@_split:NnTFn #1 {#2}
      { \prg_return_true: }
      { \prg_return_false: }
      { ... }
  }
```

but `\__prop_split:NnTFn` is non-expandable. Instead, we use `\prop_map_tokens:Nn` to compare the search key to each key in turn using `\str_if_eq:ee`, which is expandable.

```
\prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
  {
    \__prop_if_flat:NTF #1
      {
        \exp_after:wN \prop_map_tokens:Nn \exp_after:wN #1
          {
            \exp_after:wN \__prop_if_in_flat:nnn
```

```
        \exp_after:wN { \tl_to_str:n {#2} }
      }
    \prg_return_false:
  }
  {
    \exp_after:wN \__prop_get_linked:w #1 {#2}
    \use_none:n \prg_return_true: \prg_return_false:
  }
}
\cs_new:Npn \__prop_if_in_flat:nnn #1#2#3
  {
    \str_if_eq:eeT {#1} {#2}
      { \prop_map_break:n { \use_i:nn \prg_return_true: } }
  }
\prg_generate_conditional_variant:Nnn \prop_if_in:Nn
  { NV , Ne , No , c , cV , ce , co } { p , T , F , TF }
```

(*End of definition for* `\prop_if_in:NnTF` *and* `\__prop_if_in_flat:nnn`*. This function is documented on page* 218*.*)

63.8 Mapping over property lists

`\prop_map_function:NN`
`\prop_map_function:Nc`
`\prop_map_function:cN`
`\prop_map_function:cc`
`\__prop_map_function:Nw`

We first `f`-expand to flatten `#1` in case it was a linked list. The `\use_i:nnn` removes the leading `\s__prop \__prop_chk:w` of the flattened prop. The even-numbered arguments of `\__prop_map_function:Nw` are keys, hence have string catcodes, except at the end where they are `\fi: \prop_map_break:`. The `\fi:` ends the `\if_false: #⟨even⟩ \fi:` construction and we jump out of the loop. No need for any quark test.

```
\cs_new:Npn \prop_map_function:NN #1#2
  {
    \exp_last_unbraced:Nnf
      \use_i:nnn { \__prop_map_function:Nw #2 } #1
    \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
    \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
    \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
    \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
    \prg_break_point:Nn \prop_map_break: { }
  }
\cs_new:Npn \__prop_map_function:Nw #1
    \__prop_pair:wn #2 \s__prop #3
    \__prop_pair:wn #4 \s__prop #5
    \__prop_pair:wn #6 \s__prop #7
    \__prop_pair:wn #8 \s__prop #9
  {
    \if_false: #2 \fi: #1 {#2} {#3}
    \if_false: #4 \fi: #1 {#4} {#5}
    \if_false: #6 \fi: #1 {#6} {#7}
    \if_false: #8 \fi: #1 {#8} {#9}
    \__prop_map_function:Nw #1
  }
\cs_generate_variant:Nn \prop_map_function:NN { Nc , c , cc }
```

(*End of definition for* `\prop_map_function:NN` *and* `\__prop_map_function:Nw`*. This function is documented on page* 219*.*)

\prop_map_inline:Nn
\prop_map_inline:cn

Mapping in line requires a nesting level counter. Store the current definition of __prop_pair:wn, and define it anew. At the end of the loop, revert to the earlier definition. Note that besides pairs of the form __prop_pair:wn ⟨*key*⟩ \s__prop {⟨*value*⟩}, there are a leading and a trailing tokens, but both are equal to \scan_stop:, hence have no effect in such inline mapping. Such \scan_stop: could have affected ligatures if they appeared during the mapping.

```
\cs_new_protected:Npn \prop_map_inline:Nn #1#2
  {
    \cs_gset_eq:cN
      { __prop_map_ \int_use:N \g__kernel_prg_map_int :wn } \__prop_pair:wn
    \int_gincr:N \g__kernel_prg_map_int
    \cs_gset_protected:Npn \__prop_pair:wn ##1 \s__prop ##2 {#2}
    \exp_last_unbraced:Nf \use_none:nn #1
    \prg_break_point:Nn \prop_map_break:
      {
        \int_gdecr:N \g__kernel_prg_map_int
        \cs_gset_eq:Nc \__prop_pair:wn
          { __prop_map_ \int_use:N \g__kernel_prg_map_int :wn }
      }
  }
\cs_generate_variant:Nn \prop_map_inline:Nn { c }
```

(*End of definition for* \prop_map_inline:Nn. *This function is documented on page 219.*)

\prop_map_tokens:Nn
\prop_map_tokens:cn
__prop_map_tokens:nw

The mapping is very similar to \prop_map_function:NN. The odd construction \use:n {#1} allows #1 to contain any token without interfering with \prop_map_break:. The loop stops when the ⟨*key*⟩ between __prop_pair:wn and \s__prop is \fi: \prop_map_break: instead of being a string.

```
\cs_new:Npn \prop_map_tokens:Nn #1#2
  {
    \exp_last_unbraced:Nnf
      \use_i:nnn { \__prop_map_tokens:nw {#2} } #1
    \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
    \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
    \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
    \__prop_pair:wn \fi: \prop_map_break: \s__prop { }
    \prg_break_point:Nn \prop_map_break: { }
  }
\cs_new:Npn \__prop_map_tokens:nw #1
    \__prop_pair:wn #2 \s__prop #3
    \__prop_pair:wn #4 \s__prop #5
    \__prop_pair:wn #6 \s__prop #7
    \__prop_pair:wn #8 \s__prop #9
  {
    \if_false: #2 \fi: \use:n {#1} {#2} {#3}
    \if_false: #4 \fi: \use:n {#1} {#4} {#5}
    \if_false: #6 \fi: \use:n {#1} {#6} {#7}
    \if_false: #8 \fi: \use:n {#1} {#8} {#9}
    \__prop_map_tokens:nw {#1}
  }
\cs_generate_variant:Nn \prop_map_tokens:Nn { c }
```

(*End of definition for* \prop_map_tokens:Nn *and* __prop_map_tokens:nw. *This function is documented on page 219.*)

\prop_map_break:
\prop_map_break:n

The break statements are based on the general \prg_map_break:Nn.

```
\cs_new:Npn \prop_map_break:
  { \prg_map_break:Nn \prop_map_break: { } }
\cs_new:Npn \prop_map_break:n
  { \prg_map_break:Nn \prop_map_break: }
```

(*End of definition for* \prop_map_break: *and* \prop_map_break:n*. These functions are documented on page 220.*)

63.9 Uses of mapping over property lists

\prop_count:N
\prop_count:c
__prop_count:nn

Counting the key–value pairs in a property list is done using the same approach as for other count functions: turn each entry into a +1 then use integer evaluation to actually do the mathematics.

```
\cs_new:Npn \prop_count:N #1
  {
    \int_eval:n
      {
        0
        \prop_map_function:NN #1 \__prop_count:nn
      }
  }
\cs_new:Npn \__prop_count:nn #1#2 { + 1 }
\cs_generate_variant:Nn \prop_count:N { c }
```

(*End of definition for* \prop_count:N *and* __prop_count:nn*. This function is documented on page 217.*)

\prop_to_keyval:N
__prop_to_keyval_exp_after:wN
__prop_to_keyval:nn
__prop_to_keyval:nnw

Each property name and value pair will be returned in the form ␣{⟨*name*⟩}=␣{⟨*value*⟩}. As one of the main use cases for this macro is to pass the ⟨property list⟩ on to a key–value parser, we have to make sure that the behaviour is as good as possible. Using a space before the opening brace we get the correct brace stripping behaviour for most of the key–value parsers available in LaTeX. Iterate over the ⟨property list⟩ and remove the leading comma afterwards. Only the value has to be protected in __kernel_exp_not:w as the property name is always a string. After the loop the leading comma is removed by \use_none:n and afterwards __kernel_exp_not:w eventually finds the opening brace of its argument.

```
\cs_new:Npn \prop_to_keyval:N #1
  {
    \__kernel_exp_not:w
      \prop_if_empty:NTF #1
        { {} }
        {
          \exp_after:wN \exp_after:wN \exp_after:wN
          {
            \tex_expanded:D
              {
                \exp_not:N \use_none:n
                \prop_map_function:NN #1 \__prop_to_keyval:nn
              }
          }
        }
  }
```

```
\cs_new:Npn \__prop_to_keyval:nn #1#2
  { , ~ {#1} =~ { \__kernel_exp_not:w {#2} } }
```

(*End of definition for* `\prop_to_keyval:N` *and others. This function is documented on page* 217.)

63.10 Viewing property lists

`\prop_show:N`
`\prop_show:c`
`\prop_log:N`
`\prop_log:c`
`\__prop_show:NN`
`\__prop_show_finally:NNn`
`\__prop_show_prepare:w`
`\__prop_show_loop:NNw`
`\__prop_show_bad_name:NNN`
`\__prop_show_end:NNN`
`\__prop_show_loop_key:wNNN`
`\__prop_show_flat:w`
`\__prop_show_linked:w`

Experience shows one source of problems is very hard to debug: when a data structure such as a `seq` or `prop` gets corrupted. In the past, `\prop_show:N` would in some cases happily show items of such a `prop`, even though other more demanding l3prop functions would choke. It is thus best to make `\prop_show:N` check very thoroughly the structure and flag issues, even though that is very painful for linked props. Throughout the code below, we strive to remain as safe as possible, but in the explanations we only state what the arguments are when the prop is correctly formed, rather than saying at every step that various arguments can be arbitrary junk, made safe by using `\tl_to_str:n` generously.

The general `\__kernel_chk_tl_type:NnnT` checks that its first argument is a token list, and if it is, then it e-expands its second argument and compares with the contents of its first argument. Thus, within this e-expansion it is safe to use `\__prop_if_flat:NTF` to check if the prop is flat or linked. In the flat case we simply reconstruct the expected structure using `\__prop_show_flat:w`, which loops through the prop and correctly turns all keys to strings for instance. In the linked case, we use `\__prop_show_linked:w`, which ensures the form `\__prop_flatten:w \__prop` ⟨*prefix*⟩ `\s__prop {`⟨*prefix*⟩`}` ⟨*rest*⟩, where ⟨*prefix*⟩ is made into a string and ⟨*rest*⟩ cannot be a brace group or multiple tokens since `\__prop_show_linked:w` would in such cases give a different result from the original token list.

```
\cs_new_protected:Npn \prop_show:N { \__prop_show:NN \msg_show:nneeee }
\cs_generate_variant:Nn \prop_show:N { c }
\cs_new_protected:Npn \prop_log:N { \__prop_show:NN \msg_log:nneeee }
\cs_generate_variant:Nn \prop_log:N { c }
\cs_new_protected:Npn \__prop_show:NN #1#2
  {
    \__kernel_chk_tl_type:NnnT #2 { prop }
      {
        \__prop_if_flat:NTF #2
          {
            \s__prop \__prop_chk:w
            \exp_after:wN \__prop_show_flat:w #2
            \s__prop { }
            \__prop_pair:wn \q__prop_recursion_tail \s__prop { }
            \q__prop_recursion_stop
          }
          { \exp_after:wN \__prop_show_linked:w #2 \s__prop ! ? \s__prop_stop }
      }
      {
        \__prop_if_flat:NTF #2
          { \__prop_show_finally:NNn #1 #2 { flat } }
          {
            \tl_set:Nn \l__prop_internal_tl { #1 #2 }
            \exp_after:wN \__prop_show_prepare:w #2 #2
          }
      }
```

```
  }
\cs_new:Npn \__prop_show_flat:w #1 \__prop_pair:wn #2 \s__prop #3
  {
    \__prop_if_recursion_tail_stop:n {#2}
    \exp_not:N \__prop_pair:wn \tl_to_str:n {#2} \s__prop \exp_not:n { {#3} }
    \__prop_show_flat:w
  }
\cs_new:Npn \__prop_show_linked:w #1 \s__prop #2#3#4 \s__prop_stop
  {
    \exp_not:N \__prop_flatten:w
    \exp_not:c { __prop ~ \tl_to_str:n {#2} }
    \s__prop { \tl_to_str:n {#2} }
    \exp_not:n {#3}
  }
```

For flat props we are done by using `\msg_show:nneeee` or `\msg_log:nneeee`. The auxiliary `\__prop_show_finally:NNn` is eventually also used in the linked case after some more tests. To avoid having to bring along the message function and the property list, we store them into `\l__prop_internal_tl`.

```
\cs_new_protected:Npn \__prop_show_finally:NNn #1#2#3
  {
    #1 { prop } { show }
      { \token_to_str:N #2 }
      { \prop_map_function:NN #2 \msg_show_item:nn }
      {#3} { }
  }
```

For linked props, we now know they have a reasonable form so that we are calling `\__prop_show_prepare:w \__prop_flatten:w \__prop` ⟨*prefix*⟩ `\s__prop {`⟨*prefix*⟩`}` ⟨*token*⟩ ⟨*property list*⟩, and the task is to loop through the linked list and check integrity. We first set things up: the auxiliary `\__prop_tmp:w` will be in charge of checking that various tokens start with `\__prop` ⟨*prefix*⟩ (in the sense of string representations), and calling one of `\__prop_show_loop_key:wNNN`, `\__prop_show_end:NNN`, `\__prop_show_bad_name:NNN`.

```
\cs_new_protected:Npn \__prop_show_prepare:w
    \__prop_flatten:w #1 \s__prop #2#3#4
  {
    \use:e
      {
        \cs_set_nopar:Npn \exp_not:N \__prop_tmp:w
            ##1 \token_to_str:N #1 ##2 \s__prop_mark ##3 \s__prop_stop
          {
            \exp_not:N \tl_if_empty:nTF {##1}
              {
                \exp_not:N \tl_if_head_is_space:nTF {##2}
                  { \exp_not:N \exp_args:Nf \__prop_show_loop_key:wNNN }
                  { \exp_not:N \tl_if_empty:nTF }
                {##2}
              }
              { \exp_not:N \use_ii:nn }
            \__prop_show_end:NNN
            \__prop_show_bad_name:NNN
          }
      }
```

```
    \exp_last_unbraced:NNNo \__prop_show_loop:NNw #1 #4 #4
  }
```

The loop will consist of calls to `\__prop_show_loop:NNw \__prop` ⟨*prefix*⟩ ⟨*token*⟩ ⟨*expansion*⟩, where ⟨*token*⟩ is one of the items in the list, specifically the key container for ⟨key_{i-1}⟩ (starting at $i = 1$ with the property list variable itself), and ⟨*expansion*⟩ stands for the expansion of that token, which has already been checked, and takes the form ⟨*junk*⟩ `\s__prop` {⟨*value*⟩} `\__prop` ⟨*prefix*⟩ ⟨key_i⟩. Thus, the loop auxiliary receives the prefix command as `#1`, and the $(i-1)$-th and i-th key containers as `#2` and `#5`. Then `\__prop_tmp:w` checks that the name of the i-th key container is valid.

```
\cs_new_protected:Npn \__prop_show_loop:NNw #1#2 #3 \s__prop #4#5
  {
    \exp_last_two_unbraced:Noo \__prop_tmp:w
      { \token_to_str:N #5 \s__prop_mark }
      { \token_to_str:N #1 \s__prop_mark \s__prop_stop }
      #1 #2 #5
  }
```

If the i-th key container has the wrong name we get `\__prop_show_bad_name:NNN \__prop` ⟨*prefix*⟩ ⟨*previous container*⟩ ⟨*current container with bad name*⟩.

```
\cs_new_protected:Npn \__prop_show_bad_name:NNN #1#2#3
  {
    \msg_error:nneeee { prop } { bad-link }
      { \tl_tail:N \l__prop_internal_tl }
      { \token_to_str:N #2 }
      { \token_to_str:N #3 }
      { \token_to_str:N #1 }
  }
```

If the i-th key container has the name `\__prop` ⟨*prefix*⟩ (without space), it is the trailing one. We check that it is the right kind of macro to be a token list, and that it has the right contents `\use_none:n` ⟨*previous container*⟩. If so, we are done checking everything, and we display the property list using the message function and property list name stored in `\l__prop_internal_tl`. Note that we also use this `\l__prop_internal_tl` in the type argument of `\__kernel_chk_tl_type:NnnT`, to build up the name "⟨*property list*⟩ prop entry" used in error messages.

```
\cs_new_protected:Npn \__prop_show_end:NNN #1#2#3
  {
    \__kernel_chk_tl_type:NnnT #3
      { \tl_tail:N \l__prop_internal_tl prop~entry }
      { \exp_not:n { \use_none:n #2 } }
      {
        \exp_after:wN \__prop_show_finally:NNn
        \l__prop_internal_tl { linked }
      }
  }
```

If the i-th container has a name `\__prop` ⟨*prefix*⟩ ⟨*key*⟩ (with a space before the key), then we have a call to `\__prop_show_loop_key:wNNN` {⟨*key*⟩} ⟨$junk_1$⟩ ⟨$junk_2$⟩ `\__prop` ⟨*prefix*⟩ ⟨*previous container*⟩ ⟨*current container*⟩. (with an `f`-expansion to eliminate the space). The first argument is the ⟨*key*⟩ without a leading space, thanks to a judicious `f`-expansion earlier on. We check that the ⟨*current container*⟩ is a token list with the expected structure `\use_none:n` ⟨*previous container*⟩ `\__prop_pair:wn` ⟨*string*⟩ `\s__prop` {⟨*anything*⟩} ⟨*single token*⟩. The auxiliary `\__prop_show_flat:w` is reused to produce the `\__prop_pair:wn` part, and the last token is produced by `\tl_item:Nn` (we don't waste a specialized auxiliary to speed that up). If the check succeed, move on to the next item.

```
\cs_new_protected:Npn \__prop_show_loop_key:wNNN #1#2#3#4#5#6
  {
    \__kernel_chk_tl_type:NnnT #6
      { \tl_tail:N \l__prop_internal_tl prop~entry }
      {
        \exp_not:n { \use_none:n #5 }
        \exp_after:wN \__prop_show_flat:w #6 \s__prop { }
        \__prop_pair:wn \q__prop_recursion_tail \s__prop { }
        \q__prop_recursion_stop
        \tl_item:Nn #6 { -1 }
      }
      { \exp_last_unbraced:NNNo \__prop_show_loop:NNw #4 #6 #6 }
  }
```

(*End of definition for* `\prop_show:N` *and others. These functions are documented on page 220.*)

```
⟨/package⟩
```

Chapter 64

l3skip implementation

```
⟨*package⟩
⟨@@=dim⟩
```

64.1 Length primitives renamed

\if_dim:w
__dim_eval:w
__dim_eval_end:

Primitives renamed.

```
\cs_new_eq:NN \if_dim:w       \tex_ifdim:D
\cs_new_eq:NN \__dim_eval:w      \tex_dimexpr:D
\cs_new_eq:NN \__dim_eval_end:   \tex_relax:D
```

(*End of definition for* `\if_dim:w`*,* `\__dim_eval:w`*, and* `\__dim_eval_end:`*. This function is documented on page* *237*.)

64.2 Internal auxiliaries

\s__dim_mark
\s__dim_stop

Internal scan marks.

```
\scan_new:N \s__dim_mark
\scan_new:N \s__dim_stop
```

(*End of definition for* `\s__dim_mark` *and* `\s__dim_stop`*.*)

__dim_use_none_delimit_by_s_stop:w

Functions to gobble up to a scan mark.

```
\cs_new:Npn \__dim_use_none_delimit_by_s_stop:w #1 \s__dim_stop { }
```

(*End of definition for* `\__dim_use_none_delimit_by_s_stop:w`*.*)

64.3 Creating and initialising dim variables

\dim_new:N
\dim_new:c

Allocating ⟨*dim*⟩ registers ...

```
\cs_new_protected:Npn \dim_new:N #1
  {
    \__kernel_chk_if_free_cs:N #1
    \cs:w newdimen \cs_end: #1
  }
\cs_generate_variant:Nn \dim_new:N { c }
```

(*End of definition for* `\dim_new:N`. *This function is documented on page 222.*)

`\dim_const:Nn`
`\dim_const:cn`

Contrarily to integer constants, we cannot avoid using a register, even for constants. We cannot use `\dim_gset:Nn` because debugging code would complain that the constant is not a global variable. Since `\dim_const:Nn` does not need to be fast, use `\dim_eval:n` to avoid needing a debugging patch that wraps the expression in checking code.

```
\cs_new_protected:Npn \dim_const:Nn #1#2
  {
    \dim_new:N #1
    \tex_global:D #1 = \dim_eval:n {#2} \scan_stop:
  }
\cs_generate_variant:Nn \dim_const:Nn { c }
```

(*End of definition for* `\dim_const:Nn`. *This function is documented on page 222.*)

`\dim_zero:N`
`\dim_zero:c`
`\dim_gzero:N`
`\dim_gzero:c`

Reset the register to zero. Using `\c_zero_skip` deals with the case where the variable passed is incorrectly a skip (for example a LaTeX 2_ε length). Besides, these functions are then simply copied for `\skip_zero:N` and related functions.

```
\cs_new_protected:Npn \dim_zero:N #1 { #1 = \c_zero_skip }
\cs_new_protected:Npn \dim_gzero:N #1
  { \tex_global:D #1 = \c_zero_skip }
\cs_generate_variant:Nn \dim_zero:N  { c }
\cs_generate_variant:Nn \dim_gzero:N { c }
```

(*End of definition for* `\dim_zero:N` *and* `\dim_gzero:N`. *These functions are documented on page 222.*)

`\dim_zero_new:N`
`\dim_zero_new:c`
`\dim_gzero_new:N`
`\dim_gzero_new:c`

Create a register if needed, otherwise clear it.

```
\cs_new_protected:Npn \dim_zero_new:N  #1
  { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
\cs_new_protected:Npn \dim_gzero_new:N #1
  { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
\cs_generate_variant:Nn \dim_zero_new:N  { c }
\cs_generate_variant:Nn \dim_gzero_new:N { c }
```

(*End of definition for* `\dim_zero_new:N` *and* `\dim_gzero_new:N`. *These functions are documented on page 222.*)

`\dim_if_exist_p:N`
`\dim_if_exist_p:c`
`\dim_if_exist:NTF`
`\dim_if_exist:cTF`

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \dim_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End of definition for* `\dim_if_exist:NTF`. *This function is documented on page 223.*)

64.4 Setting dim variables

`\dim_set:Nn`
`\dim_set:cn`
`\dim_set:NV`
`\dim_set:cV`
`\dim_gset:Nn`
`\dim_gset:cn`
`\dim_gset:NV`
`\dim_gset:cV`

Setting dimensions is easy enough but when debugging we want both to check that the variable is correctly local/global and to wrap the expression in some code. The `\scan_stop:` deals with the case where the variable passed is a skip (for example a LaTeX 2_ε length).

```
\cs_new_protected:Npn \dim_set:Nn #1#2
  { #1 = \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
```

```
\cs_new_protected:Npn \dim_gset:Nn #1#2
  { \tex_global:D #1 = \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
\cs_generate_variant:Nn \dim_set:Nn  { NV , c , cV }
\cs_generate_variant:Nn \dim_gset:Nn { NV , c , cV }
```

(*End of definition for* `\dim_set:Nn` *and* `\dim_gset:Nn`*. These functions are documented on page 223.*)

`\dim_set_eq:NN` `\dim_set_eq:cN` `\dim_set_eq:Nc` `\dim_set_eq:cc` `\dim_gset_eq:NN` `\dim_gset_eq:cN` `\dim_gset_eq:Nc` `\dim_gset_eq:cc`

All straightforward, with a `\scan_stop:` to deal with the case where `#1` is (incorrectly) a skip.

```
\cs_new_protected:Npn \dim_set_eq:NN #1#2
  { #1 = #2 \scan_stop: }
\cs_generate_variant:Nn \dim_set_eq:NN { c , Nc , cc }
\cs_new_protected:Npn \dim_gset_eq:NN #1#2
  { \tex_global:D #1 = #2 \scan_stop: }
\cs_generate_variant:Nn \dim_gset_eq:NN { c , Nc , cc }
```

(*End of definition for* `\dim_set_eq:NN` *and* `\dim_gset_eq:NN`*. These functions are documented on page 223.*)

`\dim_add:Nn` `\dim_add:cn` `\dim_gadd:Nn` `\dim_gadd:cn` `\dim_sub:Nn` `\dim_sub:cn` `\dim_gsub:Nn` `\dim_gsub:cn`

Using `by` here would slow things down just to detect nonsensical cases such as passing `\dimen 123` as the first argument. Using `\scan_stop:` deals with skip variables. Since debugging checks that the variable is correctly local/global, the global versions cannot be defined as `\tex_global:D` followed by the local versions.

```
\cs_new_protected:Npn \dim_add:Nn #1#2
  { \tex_advance:D #1 \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
\cs_new_protected:Npn \dim_gadd:Nn #1#2
  {
    \tex_global:D \tex_advance:D #1
      \__dim_eval:w #2 \__dim_eval_end: \scan_stop:
  }
\cs_generate_variant:Nn \dim_add:Nn  { c }
\cs_generate_variant:Nn \dim_gadd:Nn { c }
\cs_new_protected:Npn \dim_sub:Nn #1#2
  { \tex_advance:D #1 - \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
\cs_new_protected:Npn \dim_gsub:Nn #1#2
  {
    \tex_global:D \tex_advance:D #1
      -\__dim_eval:w #2 \__dim_eval_end: \scan_stop:
  }
\cs_generate_variant:Nn \dim_sub:Nn  { c }
\cs_generate_variant:Nn \dim_gsub:Nn { c }
```

(*End of definition for* `\dim_add:Nn` *and others. These functions are documented on page 223.*)

64.5 Utilities for dimension calculations

`\dim_abs:n` `\__dim_abs:N` `\dim_max:nn` `\dim_min:nn` `\__dim_maxmin:wwN`

Functions for min, max, and absolute value with only one evaluation. The absolute value is evaluated by removing a leading `-` if present.

```
\cs_new:Npn \dim_abs:n #1
  {
    \exp_after:wN \__dim_abs:N
    \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
  }
```

```
\cs_new:Npn \__dim_abs:N #1
  { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
\cs_new:Npn \dim_max:nn #1#2
  {
    \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
      \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
      \dim_use:N \__dim_eval:w #2 ;
      >
    \__dim_eval_end:
  }
\cs_new:Npn \dim_min:nn #1#2
  {
    \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
      \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
      \dim_use:N \__dim_eval:w #2 ;
      <
    \__dim_eval_end:
  }
\cs_new:Npn \__dim_maxmin:wwN #1 ; #2 ; #3
  {
    \if_dim:w #1 #3 #2 ~
      #1
    \else:
      #2
    \fi:
  }
```

(*End of definition for* `\dim_abs:n` *and others. These functions are documented on page 223.*)

`\dim_ratio:nn`
`\__dim_ratio:n`

With dimension expressions, something like `10 pt * ( 5 pt / 10 pt )` does not work. Instead, the ratio part needs to be converted to an integer expression. Using `\int_value:w` forces everything into `sp`, avoiding any decimal parts.

```
\cs_new:Npn \dim_ratio:nn #1#2
  { \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
\cs_new:Npn \__dim_ratio:n #1
  { \int_value:w \__dim_eval:w (#1) \__dim_eval_end: }
```

(*End of definition for* `\dim_ratio:nn` *and* `\__dim_ratio:n`*. This function is documented on page 224.*)

64.6 Dimension expression conditionals

`\dim_compare_p:nNn`
`\dim_compare:nNnTF`

Simple comparison.

```
\prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
  {
    \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End of definition for* `\dim_compare:nNnTF`*. This function is documented on page 224.*)

`\dim_compare_p:n`
`\dim_compare:nTF`
`\__dim_compare:w`
`\__dim_compare:wNN`
`\__dim_compare_=:w`
`\__dim_compare_!:w`
`\__dim_compare_<:w`
`\__dim_compare_>:w`
`\__dim_compare_error:`

This code is adapted from the `\int_compare:nTF` function. First make sure that there is at least one relation operator, by evaluating a dimension expression with a trailing `\__dim_compare_error:`. Just like for integers, the looping auxiliary `\__dim_compare:wNN` closes a primitive conditional and opens a new one. It is actually easier to grab a dimension operand than an integer one, because once evaluated, dimensions all end with `pt` (with category other). Thus we do not need specific auxiliaries for the three "simple" relations `<`, `=`, and `>`.

```
\prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
  {
    \exp_after:wN \__dim_compare:w
    \dim_use:N \__dim_eval:w #1 \__dim_compare_error:
  }
\cs_new:Npn \__dim_compare:w #1 \__dim_compare_error:
  {
    \exp_after:wN \if_false: \exp:w \exp_end_continue_f:w
      \__dim_compare:wNN #1 ? { = \__dim_compare_end:w \else: } \s__dim_stop
  }
\exp_args:Nno \use:nn
  { \cs_new:Npn \__dim_compare:wNN #1 } { \tl_to_str:n {pt} #2#3 }
  {
      \if_meaning:w = #3
        \use:c { __dim_compare_#2:w }
      \fi:
        #1 pt \exp_stop_f:
      \prg_return_false:
      \exp_after:wN \__dim_use_none_delimit_by_s_stop:w
    \fi:
    \reverse_if:N \if_dim:w #1 pt #2
      \exp_after:wN \__dim_compare:wNN
      \dim_use:N \__dim_eval:w #3
  }
\cs_new:cpn { __dim_compare_ ! :w }
    #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
\cs_new:cpn { __dim_compare_ = :w }
    #1 \__dim_eval:w = { #1 \__dim_eval:w }
\cs_new:cpn { __dim_compare_ < :w }
    #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
\cs_new:cpn { __dim_compare_ > :w }
    #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
\cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \s__dim_stop
  { #1 \prg_return_false: \else: \prg_return_true: \fi: }
\cs_new_protected:Npn \__dim_compare_error:
  {
    \if_int_compare:w \c_zero_int \c_zero_int \fi:
    =
    \__dim_compare_error:
  }
```

(*End of definition for* `\dim_compare:nTF` *and others. This function is documented on page 225.*)

`\dim_case:nn`
`\dim_case:nnTF`
`\__dim_case:nnTF`
`\__dim_case:nw`
`\__dim_case_end:nw`

For dimension cases, the first task to fully expand the check condition. The over all idea is then much the same as for `\str_case:nnTF` as described in l3basics.

```
\cs_new:Npn \dim_case:nnTF #1
  {
    \exp:w
    \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} }
  }
\cs_new:Npn \dim_case:nnT #1#2#3
```

```
  {
    \exp:w
    \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { }
  }
\cs_new:Npn \dim_case:nnF #1#2
  {
    \exp:w
    \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { }
  }
\cs_new:Npn \dim_case:nn #1#2
  {
    \exp:w
    \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { }
  }
\cs_new:Npn \__dim_case:nnTF #1#2#3#4
  { \__dim_case:nw {#1} #2 {#1} { } \s__dim_mark {#3} \s__dim_mark {#4} \s__dim_stop }
\cs_new:Npn \__dim_case:nw #1#2#3
  {
    \dim_compare:nNnTF {#1} = {#2}
      { \__dim_case_end:nw {#3} }
      { \__dim_case:nw {#1} }
  }
\cs_new:Npn \__dim_case_end:nw #1#2#3 \s__dim_mark #4#5 \s__dim_stop
  { \exp_end: #1 #4 }
```

(*End of definition for* `\dim_case:nnTF` *and others. This function is documented on page 226.*)

64.7 Dimension expression loops

`\dim_while_do:nn`
`\dim_until_do:nn`
`\dim_do_while:nn`
`\dim_do_until:nn`

`while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```
\cs_new:Npn \dim_while_do:nn #1#2
  {
    \dim_compare:nT {#1}
      {
        #2
        \dim_while_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \dim_until_do:nn #1#2
  {
    \dim_compare:nF {#1}
      {
        #2
        \dim_until_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \dim_do_while:nn #1#2
  {
    #2
    \dim_compare:nT {#1}
      { \dim_do_while:nn {#1} {#2} }
  }
```

```
\cs_new:Npn \dim_do_until:nn #1#2
  {
    #2
    \dim_compare:nF {#1}
      { \dim_do_until:nn {#1} {#2} }
  }
```

(*End of definition for* `\dim_while_do:nn` *and others. These functions are documented on page 227.*)

`\dim_while_do:nNnn` `\dim_until_do:nNnn` `\dim_do_while:nNnn` `\dim_do_until:nNnn`

`while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```
\cs_new:Npn \dim_while_do:nNnn #1#2#3#4
  {
    \dim_compare:nNnT {#1} #2 {#3}
      {
        #4
        \dim_while_do:nNnn {#1} #2 {#3} {#4}
      }
  }
\cs_new:Npn \dim_until_do:nNnn #1#2#3#4
  {
  \dim_compare:nNnF {#1} #2 {#3}
    {
      #4
      \dim_until_do:nNnn {#1} #2 {#3} {#4}
    }
  }
\cs_new:Npn \dim_do_while:nNnn #1#2#3#4
  {
    #4
    \dim_compare:nNnT {#1} #2 {#3}
      { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
  }
\cs_new:Npn \dim_do_until:nNnn #1#2#3#4
  {
    #4
    \dim_compare:nNnF {#1} #2 {#3}
      { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
  }
```

(*End of definition for* `\dim_while_do:nNnn` *and others. These functions are documented on page 227.*)

64.8 Dimension step functions

`\dim_step_function:nnnN` `\__dim_step:wwwN` `\__dim_step:NnnnN`

Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```
\cs_new:Npn \dim_step_function:nnnN #1#2#3
  {
    \exp_after:wN \__dim_step:wwwN
    \tex_the:D \__dim_eval:w #1 \exp_after:wN ;
```

```
    \tex_the:D \__dim_eval:w #2 \exp_after:wN ;
    \tex_the:D \__dim_eval:w #3 ;
  }
\cs_new:Npn \__dim_step:wwwN #1; #2; #3; #4
  {
    \dim_compare:nNnTF {#2} > \c_zero_dim
      { \__dim_step:NnnnN > }
      {
        \dim_compare:nNnTF {#2} = \c_zero_dim
          {
            \msg_expandable_error:nnn { kernel } { zero-step } {#4}
            \use_none:nnnn
          }
          { \__dim_step:NnnnN < }
      }
      {#1} {#2} {#3} #4
  }
\cs_new:Npn \__dim_step:NnnnN #1#2#3#4#5
  {
    \dim_compare:nNnF {#2} #1 {#4}
      {
        #5 {#2}
        \exp_args:NNf \__dim_step:NnnnN
          #1 { \dim_eval:n { #2 + #3 } } {#3} {#4} #5
      }
  }
```

(*End of definition for* `\dim_step_function:nnnN`*,* `\__dim_step:wwwN`*, and* `\__dim_step:NnnnN`*. This function is documented on page 227.*)

`\dim_step_inline:nnnn`
`\dim_step_variable:nnnNn`
`\__dim_step:NNnnnn`

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\dim_step_function:nnnN`. We put a `\prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g__kernel_prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so that no breaking function recognizes this break point as its own.

```
\cs_new_protected:Npn \dim_step_inline:nnnn
  {
    \int_gincr:N \g__kernel_prg_map_int
    \exp_args:NNc \__dim_step:NNnnnn
      \cs_gset_protected:Npn
      { __dim_map_ \int_use:N \g__kernel_prg_map_int :w }
  }
\cs_new_protected:Npn \dim_step_variable:nnnNn #1#2#3#4#5
  {
    \int_gincr:N \g__kernel_prg_map_int
    \exp_args:NNc \__dim_step:NNnnnn
      \cs_gset_protected:Npe
      { __dim_map_ \int_use:N \g__kernel_prg_map_int :w }
      {#1}{#2}{#3}
      {
        \tl_set:Nn \exp_not:N #4 {##1}
        \exp_not:n {#5}
      }
```

```
  }
\cs_new_protected:Npn \__dim_step:NNnnnn #1#2#3#4#5#6
  {
    #1 #2 ##1 {#6}
    \dim_step_function:nnnN {#3} {#4} {#5} #2
    \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
  }
```

(*End of definition for* `\dim_step_inline:nnnn`*,* `\dim_step_variable:nnnNn`*, and* `\__dim_step:NNnnnn`*. These functions are documented on page* *228**.*)

64.9 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```
\cs_new:Npn \dim_eval:n #1
  { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }
```

(*End of definition for* `\dim_eval:n`*. This function is documented on page* *228**.*)

`\dim_sign:n` `\__dim_sign:Nw` See `\dim_abs:n`. Contrarily to `\int_sign:n` the case of a zero dimension cannot be distinguished from a positive dimension by looking only at the first character, since `0.2pt` and `0pt` start the same way. We need explicit comparisons. We start by distinguishing the most common case of a positive dimension.

```
\cs_new:Npn \dim_sign:n #1
  {
    \int_value:w \exp_after:wN \__dim_sign:Nw
      \dim_use:N \__dim_eval:w #1 \__dim_eval_end: ;
    \exp_stop_f:
  }
\cs_new:Npn \__dim_sign:Nw #1#2 ;
  {
    \if_dim:w #1#2 > \c_zero_dim
      1
    \else:
      \if_meaning:w - #1
        -1
      \else:
        0
      \fi:
    \fi:
  }
```

(*End of definition for* `\dim_sign:n` *and* `\__dim_sign:Nw`*. This function is documented on page* *228**.*)

`\dim_use:N` `\dim_use:c` Accessing a ⟨*dim*⟩. We hand-code the `c` variant for some speed gain.

```
\cs_new_eq:NN \dim_use:N \tex_the:D
\cs_new:Npn \dim_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(*End of definition for* `\dim_use:N`*. This function is documented on page* *228**.*)

`\dim_to_decimal:n` `\__dim_to_decimal:w` A function which comes up often enough to deserve a place in the kernel. Evaluate the dimension expression `#1` then remove the trailing `pt`. When debugging is enabled, the argument is put in parentheses as this prevents the dimension expression from terminating early and leaving extra tokens lying around. This is used a lot by low-level manipulations.

```
\cs_new:Npn \dim_to_decimal:n #1
  {
    \exp_after:wN
      \__dim_to_decimal:w \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
  }
\use:e
  {
    \cs_new:Npn \exp_not:N \__dim_to_decimal:w
      #1 . #2 \tl_to_str:n { pt }
  }
      {
        \int_compare:nNnTF {#2} > \c_zero_int
          { #1 . #2 }
          { #1 }
      }
```

(*End of definition for* `\dim_to_decimal:n` *and* `\__dim_to_decimal:w`*. This function is documented on page 228.*)

`\dim_to_fp:n`

Defined in l3fp-convert, documented here.

(*End of definition for* `\dim_to_fp:n`*. This function is documented on page 230.*)

64.10 Conversion of `dim` to other units

The conversion from `pt` or `sp` to other units is complicated by the fact that TEX's conversion to `sp` involves rounding and hard-coded ratios. In order to give re-entrant outcomes, we therefore need to do quite a bit of work: see https://github.com/latex3/latex3/issues/954 for detailed discussion. After dealing with the trivial case, we therefore have some work to do. The code to do this is contributed by Ruixi Zhang.

`\dim_to_decimal_in_sp:n`

The one easy case: the only requirement here is that we avoid an overflow.

```
\cs_new:Npn \dim_to_decimal_in_sp:n #1
  { \int_value:w \__dim_eval:w #1 \__dim_eval_end: }
```

(*End of definition for* `\dim_to_decimal_in_sp:n`*. This function is documented on page 230.*)

`\dim_to_decimal_in_bp:n`
`\dim_to_decimal_in_cc:n`
`\dim_to_decimal_in_cm:n`
`\dim_to_decimal_in_dd:n`
`\dim_to_decimal_in_in:n`
`\dim_to_decimal_in_mm:n`
`\dim_to_decimal_in_pc:n`
`\__dim_to_decimal_aux:w`

We first set up a helper macro `\__dim_tmp:w` which takes two arguments. The first argument is one of the following engine-defined units: `in`, `pc`, `cm`, `mm`, `bp`, `dd`, `cc`, `nd`, and `nc`. The second argument is $\frac{1}{2}\delta^{-1}$ in reduced fraction, where $\delta > 1$ is the engine-defined conversion factor for each unit. Note that δ must be strictly larger than 1 for the following algorithm to work.

Here is how the algorithm works: Suppose that a user inputs a non-negative dimension in a unit that has conversion factor $\delta > 1$. Then this dimension is internally represented as X sp, where $X = \lfloor N\delta \rfloor$ for some integer $N \geq 0$. We then seek a formula to express this N using X. The `\dim_to_decimal_in_<unit>:n` functions shall return the number $N/2^{16}$ in decimal. This way, we guarantee the returned decimal followed by the original unit will parse to exactly X sp.

So how do we get N from X? Well, since $X = \lfloor N\delta \rfloor$, we have $X \leq N\delta < X + 1$ and $X\delta^{-1} \leq N < (X+1)\delta^{-1}$. Let's focus on the midpoint of this bounding interval for N. The midpoint is $(X + \frac{1}{2})\delta^{-1}$. The fact $\delta > 1$ implies that the bounding interval is shorter than 1 in length. Thus, (1) midpoint $+ \frac{1}{2} > N$ and (2) midpoint $+ \frac{1}{2} < N + 1$. In other words, $N = \lfloor \text{midpoint} + \frac{1}{2} \rfloor$. As long as we can rewrite the midpoint as the result of a "scaling operation" of ε-TeX, the $\lfloor \ldots + \frac{1}{2} \rfloor$ part will follow naturally. Indeed we can: midpoint $= (2X+1) \times (\frac{1}{2}\delta^{-1})$.

Addendum: If $\delta \geq 2$, then the bounding interval for N is at most $\frac{1}{2}$ wide in length. In this case, the leftpoint $X\delta^{-1}$ suffices as $N = \lfloor X\delta^{-1} + \frac{1}{2} \rfloor$. Six out of the nine units listed above can be handled in this way, which is much simpler than using midpoint. But three remaining units have $1 < \delta < 2$; they are bp ($\delta = 7227/7200$), nd ($\delta = 685/642$), and dd ($\delta = 1238/1157$), and these three must be handled using midpoint. For consistency, we shall use the midpoint approach for all nine units.

```
\group_begin:
  \cs_set_protected:Npn \__dim_tmp:w #1#2
    {
      \cs_new:cpn { dim_to_decimal_in_ #1 :n } ##1
        {
          \exp_after:wN \__dim_to_decimal_aux:w
            \int_value:w \__dim_eval:w ##1 \__dim_eval_end: ; #2 ;
        }
    }
```

Conversions to other units are now coded. Consult the pdfTeX source for each conversion factor δ. Each factor $\frac{1}{2}\delta^{-1}$ is hand-coded for accuracy (and speed). As the units nc and nd are not supported by XeTeX or (u)pTeX, they are not included here.

```
  \__dim_tmp:w { in } {   50 /  7227 } % delta = 7227/100
  \__dim_tmp:w { pc } {    1 /    24 } % delta = 12/1
  \__dim_tmp:w { cm } {  127 /  7227 } % delta = 7227/254
  \__dim_tmp:w { mm } { 1270 /  7227 } % delta = 7227/2540
  \__dim_tmp:w { bp } {  400 /   803 } % delta = 7227/7200
  \__dim_tmp:w { dd } { 1157 /  2476 } % delta = 1238/1157
  \__dim_tmp:w { cc } { 1157 / 29712 } % delta = 14856/1157
\group_end:
```

The tokens after __dim_to_decimal_aux:w shall have the following form: <number>;<half of delta i where <number> represents the input dimension in sp unit. If <number> is positive, then #1 is its leading digit and #2 (possibly empty) is all the remaining digits; If <number> is zero, then #1 is 0_{12} and #2 is empty; If <number> is negative, then #1 is its sign $-_{12}$ and #2 is all its digits. In all three cases, #1#2 is the original <number>. We can use #1 to decide whether to use the -1 formula or the +1 formula.

```
\cs_new:Npn \__dim_to_decimal_aux:w #1#2 ; #3 ;
  {
    \dim_to_decimal:n
      {
```

We need different formulae depending on whether the user input dimension is negative or not. For negative dimension (internally represented as X sp), the formula is $(2X-1) \times (\frac{1}{2}\delta^{-1})$. For non-negative dimension, the formula is $(2X+1) \times (\frac{1}{2}\delta^{-1})$. The intermediate step doubles the dimension X. To avoid overflow, we must invoke \int_eval:n.

```
        \int_eval:n
          { ( 2 * #1#2 \if:w #1 - - \else: + \fi: 1 ) * #3 }
```

Now we append sp to finish the dimension specification.

```
        sp
      }
  }
```

(*End of definition for* \dim_to_decimal_in_bp:n *and others. These functions are documented on page* *229*.)

\dim_to_decimal_in_unit:nn

```
\cs_new:Npn \dim_to_decimal_in_unit:nn #1#2
  {
    \exp_after:wN \__dim_chk_unit:w
      \int_value:w \__dim_eval:w #2 \__dim_eval_end: ; {#1}
  }
```

(*End of definition for* `\dim_to_decimal_in_unit:nn`. *This function is documented on page 230.*)

__dim_chk_unit:w

The tokens after `\__dim_chk_unit:w` shall have the following form: `<number2>;{<dimexpr1>}`, where `<number2>` represents `<dimexpr2>` in `sp` unit. If `#1` is 0_{12}, the “unit” `<dimexpr2>` must also be zero. So we throw out a “division by zero” error message at this point. Otherwise, if `#1` is $-_{12}$, we shall negate both `<dimexpr1>` and `<dimexpr2>` for later procedures.

```
\cs_new:Npn \__dim_chk_unit:w #1#2;#3
  {
    \token_if_eq_charcode:NNTF #1 0
      { \msg_expandable_error:nn { dim } { zero-unit } }
      {
        \exp_after:wN \__dim_branch_unit:w
          \int_value:w \if:w #1 - - \fi: \__dim_eval:w #3 \exp_after:wN ;
          \int_value:w \if:w #1 - - \fi: #1#2 ;
      }
  }
```

(*End of definition for* `\__dim_chk_unit:w`.)

__dim_branch_unit:w

The tokens after `\__dim_branch_unit:w` shall have the following form: `<number1>;<number2>;`, where `<number1>` represents `<dimexpr1>` in `sp` unit (whose sign is taken care of) and `<number2>` represents the absolute value of `<dimexpr2>` in `sp` unit (which is strictly positive).

As explained, the formulae $(2X\pm1)\times(\frac{1}{2}\delta^{-1})$ work if and only if $\delta =$ `<number2>`$/65536 > 1$. This corresponds to `<dimexpr2>` strictly larger than 1 pt in absolute value. In this case, we simply call `\__dim_to_decimal_aux:w` and supply $\frac{1}{2}\delta^{-1} = 32768/$`<number2>` as `<half of delta inverse>`.

Otherwise if `<number2>` $= 65536$, then `<dimexpr2>` is 1 pt in absolute value and we call `\dim_to_decimal:n` directly.

Otherwise $0 <$ `<number2>` < 65536 and we shall proceed differently.

For unit less than 1 pt, write $n =$ `<number2>`, then $\delta = n/65536 < 1$. The midpoint formulae are not optimal. Let's go back to the inequalities $X\delta^{-1} \le N < (X+1)\delta^{-1}$. Since now $\delta < 1$, the bounding interval is wider than 1 in length. Consider the ceiling integer $M = \lceil X\delta^{-1} \rceil$, then $X\delta^{-1} \le M < (X+1)\delta^{-1}$, or equivalently $X \le M\delta < X+1$, and thus $\lfloor M\delta \rfloor = X$. The key point here is that we *don't* need to solve for N; in fact, any integer that can reproduce X (such as M) is good enough. So the algorithm goes like this: (1) Compute rounding of $X\delta^{-1}$, i.e., $M' = \lfloor X\delta^{-1} + \frac{1}{2} \rfloor$; this M' could be either M or $M-1$. (2) Check if $\lfloor M'\delta \rfloor = X$, i.e., whether our candidate M' can reproduce X. If so, then this M' is good enough; if not, then we add one to M'.

But when $0 < n < 65536$, we cannot delay the problem of overflow any more. For $X\delta^{-1} = X \times 65536/n$, where X can go up to $2^{30}-1$ and n can be as small as 1, the result is well over $2^{31}-1$ (largest integer allowed within `\numexpr`). For example, `\dim_to_decimal_in_unit:nn { \maxdimen } { 1sp }`. Here, all inputs are legal, so we should be able to output `1073741823` *without* causing arithmetic overflow.

As a workaround, let's write $X = qn + r$ with some $q \geq 0$ and $0 \leq r < n$. Then $X\delta^{-1} = 65536q + 65536r/n$, and so $M' = 65536q + \lfloor 65536r/n + \frac{1}{2} \rfloor = 65536q + R'$. Computing R' will never overflow. If this R' can reproduce r, then it is good enough; otherwise we add one to R'. In the end, we shall output $q + R'/65536$ in decimal.

Note: $q = \lfloor X/n \rfloor = \lfloor \frac{2X-n}{2n} + \frac{1}{2} \rfloor$ represents the "integer" part, while $0 \leq R' \leq 65536$ represents the "fractional" part. (Can $R' = 65536$ really happen? Didn't investigate.)

```
\cs_new:Npn \__dim_branch_unit:w #1;#2;
  {
    \int_compare:nNnTF {#2} > { 65536 }
      { \__dim_to_decimal_aux:w #1 ; 32768 / #2 ; }
      {
        \int_compare:nNnTF {#2} = { 65536 }
          { \dim_to_decimal:n { #1sp } }
          { \__dim_get_quotient:w #1 ; #2 ; }
      }
  }
```

(*End of definition for* `\__dim_branch_unit:w`.)

`\__dim_get_quotient:w`

We wish to get the quotient q via rounding of $\frac{2X-n}{2n}$. When $0 \leq X < n/2$, we have $\frac{2X-n}{2n} < 0$. So, strictly speaking, `\numexpr` performs its rounding as $\lceil \frac{2X-n}{2n} - \frac{1}{2} \rceil$, not exactly what we want. However, lucky for us, only $X = 0$ makes $\lceil \frac{2X-n}{2n} - \frac{1}{2} \rceil = -1 \neq 0$ (we want 0); all other $0 < X < n/2$ make $\lceil \frac{2X-n}{2n} - \frac{1}{2} \rceil = 0 = q$. Thus, let's filter out $X = 0$ early. If $X \neq 0$, we extract its sign and leave the sign to the back. The sign does not participate in any calculations (also the code works with positive integers only). The sign is used at the last stages when we parse the decimal output.

After `\__dim_get_quotient:w` has done its job, either we have the decimal `0`, or we have `\__dim_get_remainder:w` followed by q`;`$|X|$`;`n`;<sign of X>;`.

```
\cs_new:Npn \__dim_get_quotient:w #1#2;#3;
  {
    \token_if_eq_charcode:NNTF #1 0
      { 0 }
      {
        \token_if_eq_charcode:NNTF #1 -
          {
            \exp_after:wN \exp_after:wN \exp_after:wN \__dim_get_remainder:w
              \int_eval:n {  ( 2 * #2 - #3 ) / ( 2 * #3 ) } ;
              #2 ; #3 ; - ;
          }
          {
            \exp_after:wN \exp_after:wN \exp_after:wN \__dim_get_remainder:w
              \int_eval:n { ( 2 * #1#2 - #3 ) / ( 2 * #3 ) } ;
              #1#2 ; #3 ; ;
          }
      }
  }
```

(*End of definition for* `\__dim_get_quotient:w`.)

`\__dim_get_remainder:w`

`\__dim_get_remainder:w` does not need to read the sign. After finding the remainder r, the number $|X|$ is no longer needed. We should then have `\__dim_convert_remainder:w` followed by r`;`n`;`q`;<sign of X>;`.

```
\cs_new:Npn \__dim_get_remainder:w #1;#2;#3;
  {
    \exp_after:wN \exp_after:wN \exp_after:wN \__dim_convert_remainder:w
      \int_eval:n { #2 - #1 * #3 } ;
      #3 ; #1 ;
  }
```

(*End of definition for* `\__dim_get_remainder:w`.)

`\__dim_convert_remainder:w` This is trivial. We compute $R' = \lfloor 65536r/n + \frac{1}{2} \rfloor$, then leave `\__dim_test_candidate:w` followed by R'`;`r`;`n`;`q`;<sign of X>;`.

```
\cs_new:Npn \__dim_convert_remainder:w #1;#2;
  {
    \exp_after:wN \exp_after:wN \exp_after:wN \__dim_test_candidate:w
      \int_eval:n { #1 * 65536 / #2 } ;
      #1 ; #2 ;
  }
```

(*End of definition for* `\__dim_convert_remainder:w`.)

`\__dim_test_candidate:w` Now the fun part: We take R', r and n to test whether $r = \lfloor R'\delta \rfloor$. This is done as a dimension comparison. The left-hand side, r, is simply `r sp`. The right-hand side, $\lfloor R'\delta \rfloor$, is exactly `<R’ as decimal><dimen = n sp>`. If the result is true, then we've found R'; otherwise we add one to R'. After this step, r and n are no longer needed. We should then have `\__dim_parse_decimal:w` followed by R'`;`q`;<sign of X>;`.

```
\cs_new:Npn \__dim_test_candidate:w #1;#2;#3;
  {
    \dim_compare:nNnTF { #2sp } =
      { \dim_to_decimal:n { #1sp } \__dim_eval:w #3sp \__dim_eval_end: }
      { \__dim_parse_decimal:w #1 ; }
      {
        \__dim_parse_decimal:w \int_eval:n { #1 + 1 } ;
      }
  }
```

(*End of definition for* `\__dim_test_candidate:w`.)

`\__dim_parse_decimal:w`
`\__dim_parse_decimal_aux:w`

The Grand Finale: We sum q and $R'/65536$ together, and negate the result if necessary. These are all done expandably. If $0 < R'/65536 < 1$, the integer summation is naturally terminated at the decimal point. If $R'/65536 = 0$ (or 1?), the summation is terminated at the semicolon. The auxiliary function `\__dim_parse_decimal_aux:w` takes care of both cases.

```
\cs_new:Npn \__dim_parse_decimal:w #1;#2;#3;
  {
    \exp_after:wN \__dim_parse_decimal_aux:w
      \int_value:w #3 \int_eval:w #2 + \dim_to_decimal:n { #1sp } ;
  }
\cs_new:Npn \__dim_parse_decimal_aux:w #1 ; {#1}
```

(*End of definition for* `\__dim_parse_decimal:w` *and* `\__dim_parse_decimal_aux:w`.)

64.11 Viewing dim variables

\dim_show:N
\dim_show:c

Diagnostics.

```
\cs_new_eq:NN  \dim_show:N \__kernel_register_show:N
\cs_generate_variant:Nn \dim_show:N { c }
```

(*End of definition for* `\dim_show:N`*. This function is documented on page 230.*)

\dim_show:n

Diagnostics. We don't use the TEX primitive `\showthe` to show dimension expressions: this gives a more unified output.

```
\cs_new_protected:Npn \dim_show:n
  { \__kernel_msg_show_eval:Nn \dim_eval:n }
```

(*End of definition for* `\dim_show:n`*. This function is documented on page 230.*)

\dim_log:N
\dim_log:c
\dim_log:n

Diagnostics. Redirect output of `\dim_show:n` to the log.

```
\cs_new_eq:NN \dim_log:N \__kernel_register_log:N
\cs_new_eq:NN \dim_log:c \__kernel_register_log:c
\cs_new_protected:Npn \dim_log:n
  { \__kernel_msg_log_eval:Nn \dim_eval:n }
```

(*End of definition for* `\dim_log:N` *and* `\dim_log:n`*. These functions are documented on page 230.*)

64.12 Constant dimensions

\c_zero_dim
\c_max_dim

Constant dimensions.

```
\dim_const:Nn \c_zero_dim { 0 pt }
\dim_const:Nn \c_max_dim { 16383.99999 pt }
```

(*End of definition for* `\c_zero_dim` *and* `\c_max_dim`*. These variables are documented on page 231.*)

64.13 Scratch dimensions

\l_tmpa_dim
\l_tmpb_dim
\g_tmpa_dim
\g_tmpb_dim

We provide two local and two global scratch registers, maybe we need more or less.

```
\dim_new:N \l_tmpa_dim
\dim_new:N \l_tmpb_dim
\dim_new:N \g_tmpa_dim
\dim_new:N \g_tmpb_dim
```

(*End of definition for* `\l_tmpa_dim` *and others. These variables are documented on page 231.*)

64.14 Creating and initialising skip variables

```
⟨@@=skip⟩
```

\s__skip_stop

Internal scan marks.

```
\scan_new:N \s__skip_stop
```

(*End of definition for* `\s__skip_stop`*.*)

\skip_new:N
\skip_new:c

Allocation of a new internal registers.

```
\cs_new_protected:Npn \skip_new:N #1
  {
    \__kernel_chk_if_free_cs:N #1
    \cs:w newskip \cs_end: #1
  }
\cs_generate_variant:Nn \skip_new:N { c }
```

(*End of definition for* `\skip_new:N`*. This function is documented on page* 231*.*)

\skip_const:Nn
\skip_const:cn

Contrarily to integer constants, we cannot avoid using a register, even for constants. See `\dim_const:Nn` for why we cannot use `\skip_gset:Nn`.

```
\cs_new_protected:Npn \skip_const:Nn #1#2
  {
    \skip_new:N #1
    \tex_global:D #1 = \skip_eval:n {#2} \scan_stop:
  }
\cs_generate_variant:Nn \skip_const:Nn { c }
```

(*End of definition for* `\skip_const:Nn`*. This function is documented on page* 231*.*)

\skip_zero:N
\skip_zero:c
\skip_gzero:N
\skip_gzero:c

Reset the register to zero.

```
\cs_new_eq:NN \skip_zero:N  \dim_zero:N
\cs_new_eq:NN \skip_gzero:N \dim_gzero:N
\cs_generate_variant:Nn \skip_zero:N  { c }
\cs_generate_variant:Nn \skip_gzero:N { c }
```

(*End of definition for* `\skip_zero:N` *and* `\skip_gzero:N`*. These functions are documented on page* 231*.*)

\skip_zero_new:N
\skip_zero_new:c
\skip_gzero_new:N
\skip_gzero_new:c

Create a register if needed, otherwise clear it.

```
\cs_new_protected:Npn \skip_zero_new:N  #1
  { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
\cs_new_protected:Npn \skip_gzero_new:N #1
  { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
\cs_generate_variant:Nn \skip_zero_new:N  { c }
\cs_generate_variant:Nn \skip_gzero_new:N { c }
```

(*End of definition for* `\skip_zero_new:N` *and* `\skip_gzero_new:N`*. These functions are documented on page* 232*.*)

\skip_if_exist_p:N
\skip_if_exist_p:c
\skip_if_exist:NTF
\skip_if_exist:cTF

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \skip_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \skip_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End of definition for* `\skip_if_exist:NTF`*. This function is documented on page* 232*.*)

64.15 Setting skip variables

\skip_set:Nn
\skip_set:cn
\skip_set:NV
\skip_set:cV
\skip_gset:Nn
\skip_gset:cn
\skip_gset:NV
\skip_gset:cV

Much the same as for dimensions.

```
\cs_new_protected:Npn \skip_set:Nn #1#2
  { #1 = \tex_glueexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \skip_gset:Nn #1#2
  { \tex_global:D #1 = \tex_glueexpr:D #2 \scan_stop: }
\cs_generate_variant:Nn \skip_set:Nn  { NV , c , cV }
\cs_generate_variant:Nn \skip_gset:Nn { NV , c , cV }
```

(*End of definition for* `\skip_set:Nn` *and* `\skip_gset:Nn`*. These functions are documented on page 232.*)

\skip_set_eq:NN
\skip_set_eq:cN
\skip_set_eq:Nc
\skip_set_eq:cc
\skip_gset_eq:NN
\skip_gset_eq:cN
\skip_gset_eq:Nc
\skip_gset_eq:cc

All straightforward.

```
\cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
\cs_generate_variant:Nn \skip_set_eq:NN { c , Nc , cc }
\cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\cs_generate_variant:Nn \skip_gset_eq:NN { c , Nc , cc }
```

(*End of definition for* `\skip_set_eq:NN` *and* `\skip_gset_eq:NN`*. These functions are documented on page 232.*)

\skip_add:Nn
\skip_add:cn
\skip_gadd:Nn
\skip_gadd:cn
\skip_sub:Nn
\skip_sub:cn
\skip_gsub:Nn
\skip_gsub:cn

Using by here deals with the (incorrect) case `\skip123`.

```
\cs_new_protected:Npn \skip_add:Nn #1#2
  { \tex_advance:D #1 \tex_glueexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \skip_gadd:Nn #1#2
  { \tex_global:D \tex_advance:D #1 \tex_glueexpr:D #2 \scan_stop: }
\cs_generate_variant:Nn \skip_add:Nn  { c }
\cs_generate_variant:Nn \skip_gadd:Nn { c }
\cs_new_protected:Npn \skip_sub:Nn #1#2
  { \tex_advance:D #1 - \tex_glueexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \skip_gsub:Nn #1#2
  { \tex_global:D \tex_advance:D #1 - \tex_glueexpr:D #2 \scan_stop: }
\cs_generate_variant:Nn \skip_sub:Nn  { c }
\cs_generate_variant:Nn \skip_gsub:Nn { c }
```

(*End of definition for* `\skip_add:Nn` *and others. These functions are documented on page 232.*)

64.16 Skip expression conditionals

\skip_if_eq_p:nn
\skip_if_eq:nnTF

Comparing skips means doing two expansions to make strings, and then testing them. As a result, only equality is tested.

```
\prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
  {
    \str_if_eq:eeTF { \skip_eval:n {#1} } { \skip_eval:n {#2} }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End of definition for* `\skip_if_eq:nnTF`*. This function is documented on page 233.*)

\skip_if_finite_p:n
\skip_if_finite:nTF
__skip_if_finite:wwNw

With ε-TeX, we have an easy access to the order of infinities of the stretch and shrink components of a skip. However, to access both, we either need to evaluate the expression twice, or evaluate it, then call an auxiliary to extract both pieces of information from the result. Since we are going to need an auxiliary anyways, it is quicker to make it search for the string `fil` which characterizes infinite glue.

```
\cs_set_protected:Npn \__skip_tmp:w #1
  {
    \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
      {
        \exp_after:wN \__skip_if_finite:wwNw
        \skip_use:N \tex_glueexpr:D ##1 ; \prg_return_false:
        #1 ; \prg_return_true: \s__skip_stop
      }
    \cs_new:Npn \__skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \s__skip_stop {##3}
  }
\exp_args:No \__skip_tmp:w { \tl_to_str:n { fil } }
```

(*End of definition for* `\skip_if_finite:nTF` *and* `\__skip_if_finite:wwNw`*. This function is documented on page* *233**.*)

64.17 Using skip expressions and variables

\skip_eval:n

Evaluating a skip expression expandably.

```
\cs_new:Npn \skip_eval:n #1
  { \skip_use:N \tex_glueexpr:D #1 \scan_stop: }
```

(*End of definition for* `\skip_eval:n`*. This function is documented on page* *233**.*)

\skip_use:N
\skip_use:c

Accessing a ⟨*skip*⟩.

```
\cs_new_eq:NN \skip_use:N \dim_use:N
\cs_new_eq:NN \skip_use:c \dim_use:c
```

(*End of definition for* `\skip_use:N`*. This function is documented on page* *233**.*)

64.18 Inserting skips into the output

\skip_horizontal:N
\skip_horizontal:c
\skip_horizontal:n
\skip_vertical:N
\skip_vertical:c
\skip_vertical:n

Inserting skips.

```
\cs_new_eq:NN  \skip_horizontal:N \tex_hskip:D
\cs_new:Npn \skip_horizontal:n #1
  { \skip_horizontal:N \tex_glueexpr:D #1 \scan_stop: }
\cs_new_eq:NN  \skip_vertical:N \tex_vskip:D
\cs_new:Npn \skip_vertical:n #1
  { \skip_vertical:N \tex_glueexpr:D #1 \scan_stop: }
\cs_generate_variant:Nn \skip_horizontal:N { c }
\cs_generate_variant:Nn \skip_vertical:N { c }
```

(*End of definition for* `\skip_horizontal:N` *and others. These functions are documented on page* *234**.*)

64.19 Viewing skip variables

\skip_show:N
\skip_show:c

Diagnostics.

```
\cs_new_eq:NN  \skip_show:N \__kernel_register_show:N
\cs_generate_variant:Nn \skip_show:N { c }
```

(*End of definition for* `\skip_show:N`. *This function is documented on page 233.*)

\skip_show:n

Diagnostics. We don't use the TeX primitive `\showthe` to show skip expressions: this gives a more unified output.

```
\cs_new_protected:Npn \skip_show:n
  { \__kernel_msg_show_eval:Nn \skip_eval:n }
```

(*End of definition for* `\skip_show:n`. *This function is documented on page 233.*)

\skip_log:N
\skip_log:c
\skip_log:n

Diagnostics. Redirect output of `\skip_show:n` to the log.

```
\cs_new_eq:NN \skip_log:N \__kernel_register_log:N
\cs_new_eq:NN \skip_log:c \__kernel_register_log:c
\cs_new_protected:Npn \skip_log:n
  { \__kernel_msg_log_eval:Nn \skip_eval:n }
```

(*End of definition for* `\skip_log:N` *and* `\skip_log:n`. *These functions are documented on page 234.*)

64.20 Constant skips

\c_zero_skip
\c_max_skip

Skips with no rubber component are just dimensions but need to terminate correctly.

```
\skip_const:Nn \c_zero_skip { \c_zero_dim }
\skip_const:Nn \c_max_skip { \c_max_dim }
```

(*End of definition for* `\c_zero_skip` *and* `\c_max_skip`. *These functions are documented on page 234.*)

64.21 Scratch skips

\l_tmpa_skip
\l_tmpb_skip
\g_tmpa_skip
\g_tmpb_skip

We provide two local and two global scratch registers, maybe we need more or less.

```
\skip_new:N \l_tmpa_skip
\skip_new:N \l_tmpb_skip
\skip_new:N \g_tmpa_skip
\skip_new:N \g_tmpb_skip
```

(*End of definition for* `\l_tmpa_skip` *and others. These variables are documented on page 234.*)

64.22 Creating and initialising muskip variables

\muskip_new:N
\muskip_new:c

And then we add muskips.

```
\cs_new_protected:Npn \muskip_new:N #1
  {
    \__kernel_chk_if_free_cs:N #1
    \cs:w newmuskip \cs_end: #1
  }
\cs_generate_variant:Nn \muskip_new:N { c }
```

(*End of definition for* `\muskip_new:N`. *This function is documented on page 235.*)

\muskip_const:Nn
\muskip_const:cn

See \skip_const:Nn.

```
\cs_new_protected:Npn \muskip_const:Nn #1#2
  {
    \muskip_new:N #1
    \tex_global:D #1 = \muskip_eval:n {#2} \scan_stop:
  }
\cs_generate_variant:Nn \muskip_const:Nn { c }
```

(*End of definition for* `\muskip_const:Nn`. *This function is documented on page 235.*)

\muskip_zero:N
\muskip_zero:c
\muskip_gzero:N
\muskip_gzero:c

Reset the register to zero.

```
\cs_new_protected:Npn \muskip_zero:N #1
  { #1 = \c_zero_muskip }
\cs_new_protected:Npn \muskip_gzero:N #1
  { \tex_global:D #1 = \c_zero_muskip }
\cs_generate_variant:Nn \muskip_zero:N  { c }
\cs_generate_variant:Nn \muskip_gzero:N { c }
```

(*End of definition for* `\muskip_zero:N` *and* `\muskip_gzero:N`. *These functions are documented on page 235.*)

\muskip_zero_new:N
\muskip_zero_new:c
\muskip_gzero_new:N
\muskip_gzero_new:c

Create a register if needed, otherwise clear it.

```
\cs_new_protected:Npn \muskip_zero_new:N  #1
  { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
\cs_new_protected:Npn \muskip_gzero_new:N #1
  { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
\cs_generate_variant:Nn \muskip_zero_new:N  { c }
\cs_generate_variant:Nn \muskip_gzero_new:N { c }
```

(*End of definition for* `\muskip_zero_new:N` *and* `\muskip_gzero_new:N`. *These functions are documented on page 235.*)

\muskip_if_exist_p:N
\muskip_if_exist_p:c
\muskip_if_exist:NTF
\muskip_if_exist:cTF

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \muskip_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \muskip_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End of definition for* `\muskip_if_exist:NTF`. *This function is documented on page 235.*)

64.23 Setting muskip variables

\muskip_set:Nn
\muskip_set:cn
\muskip_set:NV
\muskip_set:cV
\muskip_gset:Nn
\muskip_gset:cn
\muskip_gset:NV
\muskip_gset:cV

This should be pretty familiar.

```
\cs_new_protected:Npn \muskip_set:Nn #1#2
  { #1 = \tex_muexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \muskip_gset:Nn #1#2
  { \tex_global:D #1 = \tex_muexpr:D #2 \scan_stop: }
\cs_generate_variant:Nn \muskip_set:Nn  { NV , c , cV }
\cs_generate_variant:Nn \muskip_gset:Nn { NV , c , cV }
```

(*End of definition for* `\muskip_set:Nn` *and* `\muskip_gset:Nn`. *These functions are documented on page 236.*)

\muskip_set_eq:NN
\muskip_set_eq:cN
\muskip_set_eq:Nc
\muskip_set_eq:cc
\muskip_gset_eq:NN
\muskip_gset_eq:cN
\muskip_gset_eq:Nc
\muskip_gset_eq:cc

All straightforward.

```
\cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
\cs_generate_variant:Nn \muskip_set_eq:NN { c , Nc , cc }
\cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\cs_generate_variant:Nn \muskip_gset_eq:NN { c , Nc , cc }
```

(*End of definition for* `\muskip_set_eq:NN` *and* `\muskip_gset_eq:NN`*. These functions are documented on page* *236*.)

\muskip_add:Nn
\muskip_add:cn
\muskip_gadd:Nn
\muskip_gadd:cn
\muskip_sub:Nn
\muskip_sub:cn
\muskip_gsub:Nn
\muskip_gsub:cn

Using by here deals with the (incorrect) case \muskip123.

```
\cs_new_protected:Npn \muskip_add:Nn #1#2
  { \tex_advance:D #1 \tex_muexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \muskip_gadd:Nn #1#2
  { \tex_global:D \tex_advance:D #1 \tex_muexpr:D #2 \scan_stop: }
\cs_generate_variant:Nn \muskip_add:Nn  { c }
\cs_generate_variant:Nn \muskip_gadd:Nn { c }
\cs_new_protected:Npn \muskip_sub:Nn #1#2
  { \tex_advance:D #1 - \tex_muexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \muskip_gsub:Nn #1#2
  { \tex_global:D \tex_advance:D #1 - \tex_muexpr:D #2 \scan_stop: }
\cs_generate_variant:Nn \muskip_sub:Nn  { c }
\cs_generate_variant:Nn \muskip_gsub:Nn { c }
```

(*End of definition for* `\muskip_add:Nn` *and others. These functions are documented on page* *235*.)

64.24 Using muskip expressions and variables

\muskip_eval:n

Evaluating a muskip expression expandably.

```
\cs_new:Npn \muskip_eval:n #1
  { \muskip_use:N \tex_muexpr:D #1 \scan_stop: }
```

(*End of definition for* `\muskip_eval:n`*. This function is documented on page* *236*.)

\muskip_use:N
\muskip_use:c

Accessing a ⟨*muskip*⟩.

```
\cs_new_eq:NN \muskip_use:N \dim_use:N
\cs_new_eq:NN \muskip_use:c \dim_use:c
```

(*End of definition for* `\muskip_use:N`*. This function is documented on page* *236*.)

64.25 Viewing muskip variables

\muskip_show:N
\muskip_show:c

Diagnostics.

```
\cs_new_eq:NN  \muskip_show:N \__kernel_register_show:N
\cs_generate_variant:Nn \muskip_show:N { c }
```

(*End of definition for* `\muskip_show:N`*. This function is documented on page* *236*.)

\muskip_show:n

Diagnostics. We don't use the TeX primitive \showthe to show muskip expressions: this gives a more unified output.

```
\cs_new_protected:Npn \muskip_show:n
  { \__kernel_msg_show_eval:Nn \muskip_eval:n }
```

(*End of definition for* `\muskip_show:n`*. This function is documented on page* *237*.)

\muskip_log:N
\muskip_log:c
\muskip_log:n

Diagnostics. Redirect output of `\muskip_show:n` to the log.

```
\cs_new_eq:NN \muskip_log:N \__kernel_register_log:N
\cs_new_eq:NN \muskip_log:c \__kernel_register_log:c
\cs_new_protected:Npn \muskip_log:n
  { \__kernel_msg_log_eval:Nn \muskip_eval:n }
```

(*End of definition for* `\muskip_log:N` *and* `\muskip_log:n`. *These functions are documented on page 237.*)

64.26 Constant muskips

\c_zero_muskip
\c_max_muskip

Constant muskips given by their value.

```
\muskip_const:Nn \c_zero_muskip { 0 mu }
\muskip_const:Nn \c_max_muskip  { 16383.99999 mu }
```

(*End of definition for* `\c_zero_muskip` *and* `\c_max_muskip`. *These functions are documented on page 237.*)

64.27 Scratch muskips

\l_tmpa_muskip
\l_tmpb_muskip
\g_tmpa_muskip
\g_tmpb_muskip

We provide two local and two global scratch registers, maybe we need more or less.

```
\muskip_new:N \l_tmpa_muskip
\muskip_new:N \l_tmpb_muskip
\muskip_new:N \g_tmpa_muskip
\muskip_new:N \g_tmpb_muskip
```

(*End of definition for* `\l_tmpa_muskip` *and others. These variables are documented on page 237.*)

```
⟨/package⟩
```

Chapter 65

l3keys implementation

```
⟨*package⟩
```

65.1 Low-level interface

The low-level key parser's implementation is based heavily on expkv. Compared to keyval it adds a number of additional "safety" requirements and allows to process the parsed list of key–value pairs in a variety of ways. The net result is that this code needs around one and a half the amount of time as keyval to parse the same list of keys. To optimise speed as far as reasonably practical, a number of lower-level approaches are taken rather than using the higher-level expl3 interfaces.

```
⟨@@=keyval⟩
```

\s__keyval_nil
\s__keyval_mark
\s__keyval_stop
\s__keyval_tail

```
\scan_new:N \s__keyval_nil
\scan_new:N \s__keyval_mark
\scan_new:N \s__keyval_stop
\scan_new:N \s__keyval_tail
```

(*End of definition for* \s__keyval_nil *and others.*)

\l__kernel_keyval_allow_blank_keys_bool

The general behavior of the l3keys module is to throw an error on blank key names. However to support the usage of \keyval_parse:nnn in the l3prop module we allow this error to be switched off temporarily and just ignore blank names.

```
\bool_new:N \l__kernel_keyval_allow_blank_keys_bool
```

(*End of definition for* \l__kernel_keyval_allow_blank_keys_bool.)

This temporary macro will be used since some of the definitions will need an active comma or equals sign. Inside of this macro #1 will be the active comma and #2 will be the active equals sign.

```
\group_begin:
  \cs_set_protected:Npn \__keyval_tmp:w #1#2
    {
```

\keyval_parse:nnn
\keyval_parse:nnV
\keyval_parse:nnv
\keyval_parse:NNn
\keyval_parse:NNV
\keyval_parse:NNv

The main function starts the first of two loops. The outer loop splits the key–value list at active commas, the inner loop will do so at other commas. The use of \s__keyval_mark here prevents loss of braces from the key argument.

```
\cs_new:Npn \keyval_parse:nnn ##1 ##2 ##3
  {
    \__kernel_exp_not:w \tex_expanded:D
      {
        {
          \__keyval_loop_active:nnw {##1} {##2}
            \s__keyval_mark ##3 #1 \s__keyval_tail #1
        }
      }
  }
\cs_new_eq:NN \keyval_parse:NNn \keyval_parse:nnn
```

(*End of definition for* `\keyval_parse:nnn` *and* `\keyval_parse:NNn`*. These functions are documented on page* *252*.)

`\__keyval_loop_active:nnw` First a fast test for the end of the loop is done, it'll gobble everything up to a `\s__keyval_tail`. The loop ending macro will gobble everything to the last comma in this definition. If the end isn't reached yet, start the second loop splitting at other commas, the next iteration of this first loop will be inserted by the end of `\__keyval_loop_other:nnw`.

```
\cs_new:Npn \__keyval_loop_active:nnw ##1 ##2 ##3 #1
  {
    \__keyval_if_recursion_tail:w ##3
      \__keyval_end_loop_active:w \s__keyval_tail
    \__keyval_loop_other:nnw {##1} {##2} ##3 , \s__keyval_tail ,
  }
```

(*End of definition for* `\__keyval_loop_active:nnw`.)

`\__keyval_split_other:w` `\__keyval_split_active:w` These two macros allow to split at the first equals sign of category 12 or 13. At the same time they also execute branching by inserting the first token following `\s__keyval_mark` that followed the equals sign. Hence they also test for the presence of such an equals sign simultaneously.

```
\cs_new:Npn \__keyval_split_other:w ##1 = ##2 \s__keyval_mark ##3
  { ##3 ##1 \s__keyval_stop \s__keyval_mark ##2 }
\cs_new:Npn \__keyval_split_active:w ##1 #2 ##2 \s__keyval_mark ##3
  { ##3 ##1 \s__keyval_stop \s__keyval_mark ##2 }
```

(*End of definition for* `\__keyval_split_other:w` *and* `\__keyval_split_active:w`.)

`\__keyval_loop_other:nnw` The second loop uses the same test for its end as the first loop, next it splits at the first active equals sign using `\__keyval_split_active:w`. The `\s__keyval_nil` prevents accidental brace stripping and acts as a delimiter in the next steps. First testing for an active equals sign will reduce the number of necessary expansion steps for the expected average use case of other equals signs and hence perform better on average.

```
\cs_new:Npn \__keyval_loop_other:nnw ##1 ##2 ##3 ,
  {
    \__keyval_if_recursion_tail:w ##3
      \__keyval_end_loop_other:w \s__keyval_tail
    \__keyval_split_active:w ##3 \s__keyval_nil
      \s__keyval_mark \__keyval_split_active_auxi:w
      #2 \s__keyval_mark \__keyval_clean_up_active:w
    {##1} {##2}
    \s__keyval_mark
  }
```

(*End of definition for* `\__keyval_loop_other:nnw`.)

__keyval_split_active_auxi:w
__keyval_split_active_auxii:w
__keyval_split_active_auxiii:w
__keyval_split_active_auxiv:w
__keyval_split_active_auxv:w

After `\__keyval_split_active:w` the following will only be called if there was at least one active equals sign in the current key–value pair. Therefore this is the execution branch for a key–value pair with an active equals sign. `##1` will be everything up to the first active equals sign. First it tests for other equals signs in the key name, which will eventually throw an error via `\__keyval_misplaced_equal_after_active_error:w`. If none was found we forward the key to `\__keyval_split_active_auxii:w`.

```
      \cs_new:Npn \__keyval_split_active_auxi:w ##1 \s__keyval_stop
        {
          \__keyval_split_other:w ##1 \s__keyval_nil
            \s__keyval_mark \__keyval_misplaced_equal_after_active_error:w
            = \s__keyval_mark \__keyval_split_active_auxii:w
        }
```

`\__keyval_split_active_auxii:w` gets the correct key name with a leading `\s__keyval_mark` as `##1`. It has to sanitise the remainder of the previous test and trims the key name which will be forwarded to `\__keyval_split_active_auxiii:w`.

```
      \cs_new:Npn \__keyval_split_active_auxii:w
          ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_after_active_error:w
          \s__keyval_stop \s__keyval_mark
          ##2 \s__keyval_nil #2 \s__keyval_mark \__keyval_clean_up_active:w
        { \__keyval_trim:nN {##1} \__keyval_split_active_auxiii:w ##2 \s__keyval_nil }
```

Next we test for a misplaced active equals sign in the value, if none is found `\__keyval_split_active_auxiv:w` will be called.

```
      \cs_new:Npn \__keyval_split_active_auxiii:w ##1 ##2 \s__keyval_nil
        {
          \__keyval_split_active:w ##2 \s__keyval_nil
            \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
            #2 \s__keyval_mark \__keyval_split_active_auxiv:w
            {##1}
        }
```

This runs the last test after sanitising the remainder of the previous one. This time test for a misplaced equals sign of category 12 in the value. Finally the last auxiliary macro will be called.

```
      \cs_new:Npn \__keyval_split_active_auxiv:w
          ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
          \s__keyval_stop \s__keyval_mark
        {
          \__keyval_split_other:w ##1 \s__keyval_nil
            \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
            = \s__keyval_mark \__keyval_split_active_auxv:w
        }
```

This last macro in this execution branch sanitises the last test, trims the value and passes it to `\__keyval_pair:nnnn`.

```
      \cs_new:Npn \__keyval_split_active_auxv:w
          ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
          \s__keyval_stop \s__keyval_mark
        { \__keyval_trim:nN { ##1 } \__keyval_pair:nnnn }
```

(*End of definition for* `\__keyval_split_active_auxi:w` *and others*.)

__keyval_clean_up_active:w

The following is the branch taken if the key–value pair doesn't contain an active equals sign. The remainder of that test will be cleaned up by `\__keyval_clean_up_active:w` which will then split at an equals sign of category other.

```
\cs_new:Npn \__keyval_clean_up_active:w
    ##1 \s__keyval_nil \s__keyval_mark \__keyval_split_active_auxi:w \s__keyval_stop
  {
    \__keyval_split_other:w ##1 \s__keyval_nil
      \s__keyval_mark \__keyval_split_other_auxi:w
      = \s__keyval_mark \__keyval_clean_up_other:w
  }
```

(*End of definition for* `\__keyval_clean_up_active:w`.)

__keyval_split_other_auxi:w
__keyval_split_other_auxii:w
__keyval_split_other_auxiii:w

This is executed if the key–value pair doesn't contain an active equals sign but at least one other. `##1` of `\__keyval_split_other_auxi:w` will contain the complete key name, which is trimmed and forwarded to the next auxiliary macro.

```
\cs_new:Npn \__keyval_split_other_auxi:w ##1 \s__keyval_stop
  { \__keyval_trim:nN { ##1 } \__keyval_split_other_auxii:w }
```

We know that the value doesn't contain misplaced active equals signs but we have to test for others. Also we need to sanitise the previous test, which is done here and not earlier to avoid superfluous argument grabbing.

```
\cs_new:Npn \__keyval_split_other_auxii:w
    ##1 ##2 \s__keyval_nil = \s__keyval_mark \__keyval_clean_up_other:w
  {
    \__keyval_split_other:w ##2 \s__keyval_nil
      \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
      = \s__keyval_mark \__keyval_split_other_auxiii:w
      { ##1 }
  }
```

`\__keyval_split_other_auxiii:w` sanitises the test for other equals signs, trims the value and forwards it to `\__keyval_pair:nnnn`.

```
\cs_new:Npn \__keyval_split_other_auxiii:w
    ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
    \s__keyval_stop \s__keyval_mark
  { \__keyval_trim:nN { ##1 } \__keyval_pair:nnnn }
```

(*End of definition for* `\__keyval_split_other_auxi:w`, `\__keyval_split_other_auxii:w`, *and* `\__keyval_split_other_auxiii:w`.)

__keyval_clean_up_other:w

`\__keyval_clean_up_other:w` is the last branch that might exist. It is called if no equals sign was found, hence the only possibilities left are a blank list element, which is to be skipped, or a lonely key. If it's no empty list element this will trim the key name and forward it to `\__keyval_key:nn`.

```
\cs_new:Npn \__keyval_clean_up_other:w
    ##1 \s__keyval_nil \s__keyval_mark \__keyval_split_other_auxi:w \s__keyval_stop \
  {
    \__keyval_if_blank:w ##1 \s__keyval_nil \s__keyval_stop \__keyval_blank_true:w
      \s__keyval_mark \s__keyval_stop
      \__keyval_trim:nN { ##1 } \__keyval_key:nn
  }
```

(*End of definition for* `\__keyval_clean_up_other:w`.)

__keyval_misplaced_equal_after_active_error:w
__keyval_misplaced_equal_in_split_error:w

All these two macros do is gobble the remainder of the current other loop execution and throw an error. Afterwards they have to insert the next loop iteration.

```
      \cs_new:Npn \__keyval_misplaced_equal_after_active_error:w
          \s__keyval_mark ##1 \s__keyval_stop \s__keyval_mark ##2 \s__keyval_nil
          = \s__keyval_mark \__keyval_split_active_auxii:w
          \s__keyval_mark ##3 \s__keyval_nil
          #2 \s__keyval_mark \__keyval_clean_up_active:w
        {
          \msg_expandable_error:nn
            { keyval } { misplaced-equals-sign }
          \__keyval_loop_other:nnw
        }
      \cs_new:Npn \__keyval_misplaced_equal_in_split_error:w
          \s__keyval_mark ##1 \s__keyval_stop \s__keyval_mark ##2 \s__keyval_nil
          ##3 \s__keyval_mark ##4 ##5
        {
          \msg_expandable_error:nn
            { keyval } { misplaced-equals-sign }
          \__keyval_loop_other:nnw
        }
```

(*End of definition for* `\__keyval_misplaced_equal_after_active_error:w` *and* `\__keyval_misplaced_equal_in_split_error:w`.)

__keyval_end_loop_other:w
__keyval_end_loop_active:w

All that's left for the parsing loops are the macros which end the recursion. Both just gobble the remaining tokens of the respective loop including the next recursion call. `\__keyval_end_loop_other:w` also has to insert the next iteration of the active loop.

```
      \cs_new:Npn \__keyval_end_loop_other:w
          \s__keyval_tail
          \__keyval_split_active:w
          \s__keyval_mark \s__keyval_tail
          \s__keyval_nil \s__keyval_mark
          \__keyval_split_active_auxi:w
          #2 \s__keyval_mark \__keyval_clean_up_active:w
        { \__keyval_loop_active:nnw }
      \cs_new:Npn \__keyval_end_loop_active:w
          \s__keyval_tail
          \__keyval_loop_other:nnw ##1 \s__keyval_mark \s__keyval_tail , \s__keyval_tail ,
        { }
```

(*End of definition for* `\__keyval_end_loop_other:w` *and* `\__keyval_end_loop_active:w`.)

The parsing loops are done, so here ends the definition of `\__keyval_tmp:w`, which will finally set up the macros.

```
    }
  \char_set_catcode_active:n { `\, }
  \char_set_catcode_active:n { `\= }
  \__keyval_tmp:w , =
\group_end:
\cs_generate_variant:Nn \keyval_parse:NNn { NNV , NNv }
\cs_generate_variant:Nn \keyval_parse:nnn { nnV , nnv }
```

__keyval_pair:nnnn
__keyval_key:nn

These macros will be called on the parsed keys and values of the key–value list. All arguments are completely trimmed. They test for blank key names and call the functions passed to \keyval_parse:nnn inside of \exp_not:n with the correct arguments. Afterwards they insert the next iteration of the other loop.

```
\group_begin:
  \cs_set_protected:Npn \__keyval_tmp:w #1#2
    {
      \cs_new:Npn \__keyval_pair:nnnn ##1 ##2 ##3 ##4
        {
          \__keyval_if_blank:w \s__keyval_mark ##2 \s__keyval_nil \s__keyval_stop \__keyval
            \s__keyval_mark \s__keyval_stop
          #1
          \exp_not:n { ##4 {##2} {##1} }
          #2
          \__keyval_loop_other:nnw {##3} {##4}
        }
      \cs_new:Npn \__keyval_key:nn ##1 ##2
        {
          \__keyval_if_blank:w \s__keyval_mark ##1 \s__keyval_nil \s__keyval_stop \__keyval
            \s__keyval_mark \s__keyval_stop
          #1
          \exp_not:n { ##2 {##1} }
          #2
          \__keyval_loop_other:nnw {##2}
        }
    }
  \__keyval_tmp:w { } { }
\group_end:
```

(*End of definition for* `\__keyval_pair:nnnn` *and* `\__keyval_key:nn`.)

__keyval_if_empty:w
__keyval_if_blank:w
__keyval_if_recursion_tail:w

All these tests work by gobbling tokens until a certain combination is met, which makes them pretty fast. The test for a blank argument should be called with an arbitrary token following the argument. Each of these utilize the fact that the argument will contain a leading \s__keyval_mark.

```
\cs_new:Npn \__keyval_if_empty:w #1 \s__keyval_mark \s__keyval_stop { }
\cs_new:Npn \__keyval_if_blank:w \s__keyval_mark #1 { \__keyval_if_empty:w \s__keyval_mark
\cs_new:Npn \__keyval_if_recursion_tail:w \s__keyval_mark #1 \s__keyval_tail { }
```

(*End of definition for* `\__keyval_if_empty:w`, `\__keyval_if_blank:w`, *and* `\__keyval_if_recursion_-tail:w`.)

__keyval_blank_true:w
__keyval_blank_key_error:w

These macros will be called if the tests above didn't gobble them, they execute the branching.

```
\cs_new:Npn \__keyval_blank_true:w \s__keyval_mark \s__keyval_stop \__keyval_trim:nN #1 \__
  { \__keyval_loop_other:nnw }
\cs_new:Npn \__keyval_blank_key_error:w \s__keyval_mark \s__keyval_stop #1 \__keyval_loop_o
  {
    \bool_if:NTF \l__kernel_keyval_allow_blank_keys_bool
      { #1 }
      { \msg_expandable_error:nn { keyval } { blank-key-name } }
    \__keyval_loop_other:nnw
  }
```

(*End of definition for* `\__keyval_blank_true:w` *and* `\__keyval_blank_key_error:w`.)

Two messages for the low level parsing system.

```
\msg_new:nnn { keyval } { misplaced-equals-sign }
  { Misplaced~'='~in~key-value~input~\msg_line_context: }
\msg_new:nnn { keyval } { blank-key-name }
  { Blank~key~name~in~key-value~input~\msg_line_context: }
\prop_gput:Nnn \g_msg_module_name_prop { keyval } { LaTeX }
\prop_gput:Nnn \g_msg_module_type_prop { keyval } { }
```

__keyval_trim:nN
__keyval_trim_auxi:w
__keyval_trim_auxii:w
__keyval_trim_auxiii:w
__keyval_trim_auxiv:w

And an adapted version of `\__tl_trim_spaces:nn` which is a bit faster for our use case, as it can strip the braces at the end. This is pretty much the same concept, so I won't comment on it here. The speed gain by using this instead of `\tl_trim_spaces_apply:nN` is about 10 % of the total time for `\keyval_parse:NNn` with one key and one key–value pair, so I think it's worth it.

```
\group_begin:
  \cs_set_protected:Npn \__keyval_tmp:w #1
    {
      \cs_new:Npn \__keyval_trim:nN ##1
        {
          \__keyval_trim_auxi:w
            ##1
            \s__keyval_nil
            \s__keyval_mark #1 { }
            \s__keyval_mark \__keyval_trim_auxii:w
            \__keyval_trim_auxiii:w
            #1 \s__keyval_nil
            \__keyval_trim_auxiv:w
        }
      \cs_new:Npn \__keyval_trim_auxi:w ##1 \s__keyval_mark #1 ##2 \s__keyval_mark ##3
        {
          ##3
          \__keyval_trim_auxi:w
          \s__keyval_mark
          ##2
          \s__keyval_mark #1 {##1}
        }
      \cs_new:Npn \__keyval_trim_auxii:w \__keyval_trim_auxi:w \s__keyval_mark \s__keyval_m
        {
          \__keyval_trim_auxiii:w
          ##1
        }
      \cs_new:Npn \__keyval_trim_auxiii:w ##1 #1 \s__keyval_nil ##2
        {
          ##2
          ##1 \s__keyval_nil
          \__keyval_trim_auxiii:w
        }
```

This is the one macro which differs from the original definition.

```
      \cs_new:Npn \__keyval_trim_auxiv:w
          \s__keyval_mark ##1 \s__keyval_nil
          \__keyval_trim_auxiii:w \s__keyval_nil \__keyval_trim_auxiii:w
          ##2
        { ##2 { ##1 } }
    }
  \__keyval_tmp:w { ~ }
```

```
\group_end:
```

(*End of definition for* `\__keyval_trim:nN` *and others.*)

65.2 Constants and variables

```
⟨@@=keys⟩
```

`\c__keys_code_root_str` `\c__keys_check_root_str` `\c__keys_default_root_str` `\c__keys_groups_root_str` `\c__keys_inherit_root_str` `\c__keys_type_root_str`

Various storage areas for the different data which make up keys.

```
\str_const:Nn \c__keys_code_root_str     { key~code~>~ }
\str_const:Nn \c__keys_check_root_str    { key~check~>~ }
\str_const:Nn \c__keys_default_root_str  { key~default~>~ }
\str_const:Nn \c__keys_groups_root_str   { key~groups~>~ }
\str_const:Nn \c__keys_inherit_root_str  { key~inherit~>~ }
\str_const:Nn \c__keys_type_root_str     { key~type~>~ }
```

(*End of definition for* `\c__keys_code_root_str` *and others.*)

`\c__keys_props_root_str`

The prefix for storing properties.

```
\str_const:Nn \c__keys_props_root_str { key~prop~>~ }
```

(*End of definition for* `\c__keys_props_root_str`*.*)

`\l_keys_choice_int` `\l_keys_choice_tl`

Publicly accessible data on which choice is being used when several are generated as a set.

```
\int_new:N \l_keys_choice_int
\tl_new:N \l_keys_choice_tl
```

(*End of definition for* `\l_keys_choice_int` *and* `\l_keys_choice_tl`*. These variables are documented on page 245.*)

`\l__keys_groups_clist`

Used for storing and recovering the list of groups which apply to a key: set as a comma list but at one point we have to use this for a token list recovery.

```
\clist_new:N \l__keys_groups_clist
```

(*End of definition for* `\l__keys_groups_clist`*.*)

`\l__keys_inherit_clist`

For normalisation.

```
\clist_new:N \l__keys_inherit_clist
```

(*End of definition for* `\l__keys_inherit_clist`*.*)

`\l_keys_key_str`

The name of a key itself: needed when setting keys.

```
\str_new:N \l_keys_key_str
```

(*End of definition for* `\l_keys_key_str`*. This variable is documented on page 248.*)

`\l_keys_key_tl`

The `tl` version is deprecated but has to be handled manually.

```
\tl_new:N \l_keys_key_tl
```

(*End of definition for* `\l_keys_key_tl`*.*)

`\l__keys_module_str`

The module for an entire set of keys.

```
\str_new:N \l__keys_module_str
```

(*End of definition for* `\l__keys_module_str`*.*)

`\l__keys_no_value_bool` A marker is needed internally to show if only a key or a key plus a value was seen: this is recorded here.

```
\bool_new:N \l__keys_no_value_bool
```

(*End of definition for* `\l__keys_no_value_bool`*.*)

`\l__keys_only_known_bool` Used to track if only "known" keys are being set.

```
\bool_new:N \l__keys_only_known_bool
```

(*End of definition for* `\l__keys_only_known_bool`*.*)

`\l_keys_path_str` The "path" of the current key is stored here: this is available to the programmer and so is public.

```
\str_new:N \l_keys_path_str
```

(*End of definition for* `\l_keys_path_str`*. This variable is documented on page 248.*)

`\l_keys_path_tl` The older version is deprecated but has to be handled manually.

```
\tl_new:N \l_keys_path_tl
```

(*End of definition for* `\l_keys_path_tl`*.*)

`\l__keys_inherit_str`

```
\str_new:N \l__keys_inherit_str
```

(*End of definition for* `\l__keys_inherit_str`*.*)

`\l__keys_relative_tl` The relative path for passing keys back to the user. As this can be explicitly no-value, it must be a token list.

```
\tl_new:N \l__keys_relative_tl
\tl_set:Nn \l__keys_relative_tl { \q__keys_no_value }
```

(*End of definition for* `\l__keys_relative_tl`*.*)

`\l__keys_property_str` The "property" begin set for a key at definition time is stored here.

```
\str_new:N \l__keys_property_str
```

(*End of definition for* `\l__keys_property_str`*.*)

`\l__keys_selective_bool`
`\l__keys_exclude_bool` Two booleans for using key groups: one to indicate that "selective" setting is active, a second to specify which type ("opt-in" or "opt-out").

```
\bool_new:N \l__keys_selective_bool
\bool_new:N \l__keys_exclude_bool
```

(*End of definition for* `\l__keys_selective_bool` *and* `\l__keys_exclude_bool`*.*)

`\l__keys_selective_clist` The list of key groups being filtered in or out during selective setting.

```
\clist_new:N \l__keys_selective_clist
```

(*End of definition for* `\l__keys_selective_clist`*.*)

`\l__keys_tmp_clist` Scratch space used as a data dump.

```
\clist_new:N \l__keys_tmp_clist
```

(*End of definition for* `\l__keys_tmp_clist`*.*)

\l__keys_unused_clist

Used when setting only some keys to store those left over.

```
\clist_new:N \l__keys_unused_clist
```

(*End of definition for* `\l__keys_unused_clist`.)

\l_keys_value_tl

The value given for a key: may be empty if no value was given.

```
\tl_new:N \l_keys_value_tl
```

(*End of definition for* `\l_keys_value_tl`. *This variable is documented on page 248.*)

\l__keys_tmp_bool
\l__keys_tmpa_tl
\l__keys_tmpb_tl

Scratch space.

```
\bool_new:N \l__keys_tmp_bool
\tl_new:N \l__keys_tmpa_tl
\tl_new:N \l__keys_tmpb_tl
```

(*End of definition for* `\l__keys_tmp_bool`, `\l__keys_tmpa_tl`, *and* `\l__keys_tmpb_tl`.)

\l__keys_precompile_bool
\l__keys_precompile_tl

For digesting keys.

```
\bool_new:N \l__keys_precompile_bool
\tl_new:N \l__keys_precompile_tl
```

(*End of definition for* `\l__keys_precompile_bool` *and* `\l__keys_precompile_tl`.)

\l_keys_usage_load_prop
\l_keys_usage_preamble_prop

Global data for document-level information.

```
\prop_new:N \l_keys_usage_load_prop
\prop_new:N \l_keys_usage_preamble_prop
```

(*End of definition for* `\l_keys_usage_load_prop` *and* `\l_keys_usage_preamble_prop`. *These variables are documented on page 247.*)

65.2.1 Internal auxiliaries

\s__keys_nil
\s__keys_mark
\s__keys_stop

Internal scan marks.

```
\scan_new:N \s__keys_nil
\scan_new:N \s__keys_mark
\scan_new:N \s__keys_stop
```

(*End of definition for* `\s__keys_nil`, `\s__keys_mark`, *and* `\s__keys_stop`.)

\q__keys_no_value

Internal quarks.

```
\quark_new:N \q__keys_no_value
```

(*End of definition for* `\q__keys_no_value`.)

__keys_quark_if_no_value_p:N
__keys_quark_if_no_value:NTF

Branching quark conditional.

```
\__kernel_quark_new_conditional:Nn \__keys_quark_if_no_value:N { TF }
```

(*End of definition for* `\__keys_quark_if_no_value:NTF`.)

__keys_precompile:n

An auxiliary to allow cleaner showing of code.

```
\cs_new_protected:Npn \__keys_precompile:n #1
  {
    \bool_if:NTF \l__keys_precompile_bool
      { \tl_put_right:Nn \l__keys_precompile_tl }
      { \use:n }
        {#1}
  }
```

(*End of definition for* __keys_precompile:n.)

__keys_cs_undefine:c

Local version of \cs_undefine:c to avoid sprinkling \tex_undefined:D everywhere.

```
\cs_new_protected:Npn \__keys_cs_undefine:c #1
  {
    \if_cs_exist:w #1 \cs_end:
    \else:
      \use_i:nnnn
    \fi:
    \cs_set_eq:cN {#1} \tex_undefined:D
  }
```

(*End of definition for* __keys_cs_undefine:c.)

65.3 The key defining mechanism

\keys_define:nn
\keys_define:ne
\keys_define:nx
__keys_define:nnn
__keys_define:onn

The public function for definitions is just a wrapper for the lower level mechanism, more or less. The outer function is designed to keep a track of the current module, to allow safe nesting. The module is set removing any leading / (which is not needed here).

```
\cs_new_protected:Npn \keys_define:nn
  { \__keys_define:onn \l__keys_module_str }
\cs_generate_variant:Nn \keys_define:nn { ne , nx }
\cs_new_protected:Npn \__keys_define:nnn #1#2#3
  {
    \str_set:Ne \l__keys_module_str { \__keys_trim_spaces:n {#2} }
    \keyval_parse:NNn \__keys_define:n \__keys_define:nn {#3}
    \str_set:Nn \l__keys_module_str {#1}
  }
\cs_generate_variant:Nn \__keys_define:nnn { o }
```

(*End of definition for* \keys_define:nn *and* __keys_define:nnn*. This function is documented on page 239.*)

__keys_define:n
__keys_define:nn
__keys_define_aux:nn

The outer functions here record whether a value was given and then converge on a common internal mechanism. There is first a search for a property in the current key name, then a check to make sure it is known before the code hands off to the next step.

```
\cs_new_protected:Npn \__keys_define:n #1
  {
    \bool_set_true:N \l__keys_no_value_bool
    \__keys_define_aux:nn {#1} { }
  }
\cs_new_protected:Npn \__keys_define:nn #1#2
  {
    \bool_set_false:N \l__keys_no_value_bool
    \__keys_define_aux:nn {#1} {#2}
  }
\cs_new_protected:Npn \__keys_define_aux:nn #1#2
  {
    \__keys_property_find:n {#1}
    \cs_if_exist:cTF { \c__keys_props_root_str \l__keys_property_str }
      { \__keys_define_code:n {#2} }
      {
        \str_if_empty:NF \l__keys_property_str
```

```
          {
            \msg_error:nnee { keys } { property-unknown }
              \l__keys_property_str \l_keys_path_str
          }
      }
  }
```

(*End of definition for* `\__keys_define:n`*,* `\__keys_define:nn`*, and* `\__keys_define_aux:nn`*.*)

`\__keys_property_find:n`
`\__keys_property_find_auxi:w`
`\__keys_property_find_auxii:w`
`\__keys_property_find_auxiii:w`
`\__keys_property_find_auxiv:w`
`\__keys_property_find_err:w`

Searching for a property means finding the last `.` in the input, and storing the text before and after it. Everything is first turned into strings, so there is no problem using `\cs_set_nopar:Npe` instead of `\str_set:Ne` to set `\l_keys_path_str`. To gain further speed, brace tricks are used and `\__keys_property_find_auxiv:w` is defined as expandable. Since spaces will already be trimmed from the module we can omit it from the argument to `\__keys_trim_spaces:n`.

```
\cs_new_protected:Npn \__keys_property_find:n #1
  {
    \exp_after:wN \__keys_property_find_auxi:w \tl_to_str:n {#1}
      \s__keys_nil \__keys_property_find_auxii:w
      . \s__keys_nil \__keys_property_find_err:w
  }
\cs_new:Npn \__keys_property_find_auxi:w #1 . #2 \s__keys_nil #3
  {
    #3 #1 \s__keys_mark #2 \s__keys_nil #3
  }
\cs_new_protected:Npn \__keys_property_find_auxii:w
    #1 \s__keys_mark #2 \s__keys_nil \__keys_property_find_auxii:w . \s__keys_nil
    \__keys_property_find_err:w
  {
    \cs_set_nopar:Npe \l_keys_path_str
      {
        \str_if_empty:NF \l__keys_module_str { \l__keys_module_str / }
        \exp_after:wN \__keys_trim_spaces:n \tex_expanded:D {{
        #1
        \if_false: }}} \fi:
        \__keys_property_find_auxi:w #2 \s__keys_nil \__keys_property_find_auxiii:w
          . \s__keys_nil \__keys_property_find_auxiv:w
  }
\cs_new:Npn \__keys_property_find_auxiii:w #1 \s__keys_mark #2 . #3 \s__keys_nil #4
  {
    . #1 #4 #2 \s__keys_mark #3 \s__keys_nil #4
  }
\cs_new:Npn \__keys_property_find_auxiv:w
    #1 \s__keys_nil \__keys_property_find_auxiii:w
    \s__keys_mark \s__keys_nil \__keys_property_find_auxiv:w
  {
    \if_false: {{{ \fi: }}}
    \cs_set_nopar:Npe \l__keys_property_str { . #1 }
    \tl_set_eq:NN \l_keys_path_tl \l_keys_path_str
  }
\cs_new_protected:Npn \__keys_property_find_err:w
    #1 \s__keys_nil #2 \__keys_property_find_err:w
  {
    \str_clear:N \l__keys_property_str
```

```
    \msg_error:nnn { keys } { no-property } {#1}
  }
```

(*End of definition for* `\__keys_property_find:n` *and others.*)

`\__keys_define_code:n`
`\__keys_define_code:nnn`
`\__keys_define_code:w`

Two possible cases. If there is a value for the key, then just use the function. If not, then a check to make sure there is no need for a value with the property. If there should be one then complain, otherwise execute it. For a LaTeX 2_ε property like `.code` which doesn't contain a `:`, treat it as having arity 1 and pass the (empty) value to it.

```
\cs_new_protected:Npn \__keys_define_code:n #1
  {
    \bool_if:NTF \l__keys_no_value_bool
      {
        \__keys_define_code:nnn
          { \use:c { \c__keys_props_root_str \l__keys_property_str } {#1} }
          { \use:c { \c__keys_props_root_str \l__keys_property_str } }
          {
            \msg_error:nnee { keys } { property-requires-value }
              \l__keys_property_str \l_keys_path_str
          }
      }
      { \use:c { \c__keys_props_root_str \l__keys_property_str } {#1} }
  }
\cs_new:Npe \__keys_define_code:nnn
  {
    \exp_not:N \exp_after:wN \exp_not:N \__keys_define_code:w
      \exp_not:N \l__keys_property_str
      \c_colon_str \c_colon_str
      \exp_not:N \s__keys_stop
  }
\use:e
  {
    \cs_new:Npn \exp_not:N \__keys_define_code:w
      #1 \c_colon_str #2 \c_colon_str #3 \exp_not:N \s__keys_stop
  }
    {
      \tl_if_empty:nTF {#3}
        { \use_i:nnn }
        {
          \tl_if_empty:nTF {#2}
            { \use_ii:nnn }
            { \use_iii:nnn }
        }
    }
```

(*End of definition for* `\__keys_define_code:n`*,* `\__keys_define_code:nnn`*, and* `\__keys_define_code:w`*.*)

65.4 Turning properties into actions

`\__keys_bool_set:Nn`
`\__keys_bool_set:cn`
`\__keys_bool_set_inverse:Nn`
`\__keys_bool_set_inverse:cn`
`\__keys_bool_set:Nnnn`

Boolean keys are really just choices, but all done by hand. The second argument here is the scope: either empty or `g` for global.

```
\cs_new_protected:Npn \__keys_bool_set:Nn #1#2
```

```
  { \__keys_bool_set:Nnnn #1 {#2} { true } { false } }
\cs_generate_variant:Nn \__keys_bool_set:Nn { c }
\cs_new_protected:Npn \__keys_bool_set_inverse:Nn #1#2
  { \__keys_bool_set:Nnnn #1 {#2} { false } { true } }
\cs_generate_variant:Nn \__keys_bool_set_inverse:Nn { c }
\cs_new_protected:Npn \__keys_bool_set:Nnnn #1#2#3#4
  {
    \bool_if_exist:NF #1 { \bool_new:N #1 }
    \__keys_choice_make:
    \__keys_cmd_set:ne { \l_keys_path_str / true }
      { \exp_not:c { bool_ #2 set_ #3 :N } \exp_not:N #1 }
    \__keys_cmd_set:ne { \l_keys_path_str / false }
      { \exp_not:c { bool_ #2 set_ #4 :N } \exp_not:N #1 }
    \__keys_cmd_set_direct:nn { \l_keys_path_str / unknown }
      {
        \msg_error:nne { keys } { boolean-values-only }
          \l_keys_path_str
      }
    \__keys_default_set:n { true }
  }
\cs_generate_variant:Nn \__keys_bool_set:Nn { c }
```

(*End of definition for* `\__keys_bool_set:Nn`*,* `\__keys_bool_set_inverse:Nn`*, and* `\__keys_bool_set:Nnnn`*.*)

`\__keys_choice_make:` `\__keys_multichoice_make:` `\__keys_choice_make:N` `\__keys_choice_make_aux:N`

To make a choice from a key, two steps: set the code, and set the unknown key. As multichoices and choices are essentially the same bar one function, the code is given together.

```
\cs_new_protected:Npn \__keys_choice_make:
  { \__keys_choice_make:N \__keys_choice_find:n }
\cs_new_protected:Npn \__keys_multichoice_make:
  { \__keys_choice_make:N \__keys_multichoice_find:n }
\cs_new_protected:Npn \__keys_choice_make:N #1
  {
    \cs_if_exist:cTF
      { \c__keys_type_root_str \__keys_parent:o \l_keys_path_str }
      {
        \str_if_eq:vnTF
          { \c__keys_type_root_str \__keys_parent:o \l_keys_path_str }
          { choice }
          {
            \msg_error:nnee { keys } { nested-choice-key }
              \l_keys_path_tl { \__keys_parent:o \l_keys_path_str }
          }
          { \__keys_choice_make_aux:N #1 }
      }
      { \__keys_choice_make_aux:N #1 }
  }
\cs_new_protected:Npn \__keys_choice_make_aux:N #1
  {
    \cs_set_nopar:cpn { \c__keys_type_root_str \l_keys_path_str }
      { choice }
    \__keys_cmd_set_direct:nn \l_keys_path_str { #1 {##1} }
    \__keys_cmd_set_direct:nn { \l_keys_path_str / unknown }
```

```
      {
        \msg_error:nnee { keys } { choice-unknown }
          \l_keys_path_str {##1}
      }
  }
```

(*End of definition for* `\__keys_choice_make:` *and others.*)

`\__keys_choices_make:nn`
`\__keys_multichoices_make:nn`
`\__keys_choices_make:Nnn`

Auto-generating choices means setting up the root key as a choice, then defining each choice in turn.

```
\cs_new_protected:Npn \__keys_choices_make:nn
  { \__keys_choices_make:Nnn \__keys_choice_make: }
\cs_new_protected:Npn \__keys_multichoices_make:nn
  { \__keys_choices_make:Nnn \__keys_multichoice_make: }
\cs_new_protected:Npn \__keys_choices_make:Nnn #1#2#3
  {
    #1
    \int_zero:N \l_keys_choice_int
    \clist_map_inline:nn {#2}
      {
        \int_incr:N \l_keys_choice_int
        \__keys_cmd_set:ne
          { \l_keys_path_str / \__keys_trim_spaces:n {##1} }
          {
            \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
            \int_set:Nn \exp_not:N \l_keys_choice_int
              { \int_use:N \l_keys_choice_int }
            \exp_not:n {#3}
          }
      }
  }
```

(*End of definition for* `\__keys_choices_make:nn`*,* `\__keys_multichoices_make:nn`*, and* `\__keys_choices_make:Nnn`*.*)

`\__keys_cmd_set:nn`
`\__keys_cmd_set:Vn`
`\__keys_cmd_set:ne`
`\__keys_cmd_set:Vo`
`\__keys_cmd_set_direct:nn`

Setting the code for a key first logs if appropriate that we are defining a new key, then saves the code.

```
\cs_new_protected:Npn \__keys_cmd_set:nn #1#2
  {  \__keys_cmd_set_direct:nn {#1} { \__keys_precompile:n {#2} } }
\cs_generate_variant:Nn \__keys_cmd_set:nn { ne , Vn , Vo }
\cs_new_protected:Npn \__keys_cmd_set_direct:nn #1#2
  { \cs_set_protected:cpn { \c__keys_code_root_str #1 } ##1 {#2} }
```

(*End of definition for* `\__keys_cmd_set:nn` *and* `\__keys_cmd_set_direct:nn`*.*)

`\__keys_cs_set:NNpn`
`\__keys_cs_set:Ncpn`

Creating control sequences is a bit more tricky than other cases as we need to pick up the `p` argument. To make the internals look clearer, the trailing `n` argument here is just for appearance.

```
\cs_new_protected:Npn \__keys_cs_set:NNpn #1#2#3#
  {
    \cs_set_protected:cpe { \c__keys_code_root_str \l_keys_path_str } ##1
      {
        \__keys_precompile:n
          { #1 \exp_not:N #2 \exp_not:n {#3} {##1} }
```

```
      }
    \use_none:n
  }
\cs_generate_variant:Nn \__keys_cs_set:NNpn { Nc }
```

(*End of definition for* `\__keys_cs_set:NNpn`.)

`\__keys_default_set:n` Setting a default value is easy. These are stored using `\cs_set_nopar:cpe` as this avoids any worries about whether a token list exists.

```
\cs_new_protected:Npn \__keys_default_set:n #1
  {
    \tl_if_empty:nTF {#1}
      {
        \__keys_cs_undefine:c
          { \c__keys_default_root_str \l_keys_path_str }
      }
      {
        \cs_set_nopar:cpe
          { \c__keys_default_root_str \l_keys_path_str }
          { \exp_not:n {#1} }
        \__keys_value_requirement:nn { required } { false }
      }
  }
```

(*End of definition for* `\__keys_default_set:n`.)

`\__keys_groups_set:n` Assigning a key to one or more groups uses comma lists. As the list of groups only exists if there is anything to do, the setting is done using a scratch list. For the usual grouping reasons we use the low-level approach to undefining a list. We also use the low-level approach for the other case to avoid tripping up the `check-declarations` code.

```
\cs_new_protected:Npn \__keys_groups_set:n #1
  {
    \clist_set:Ne \l__keys_groups_clist { \tl_to_str:n {#1} }
    \clist_if_empty:NTF \l__keys_groups_clist
      {
        \__keys_cs_undefine:c
          { \c__keys_groups_root_str \l_keys_path_str }
      }
      {
        \cs_set_eq:cN { \c__keys_groups_root_str \l_keys_path_str }
          \l__keys_groups_clist
      }
  }
```

(*End of definition for* `\__keys_groups_set:n`.)

`\__keys_inherit:n` Inheritance means ignoring anything already said about the key: zap the lot and set up.

```
\cs_new_protected:Npn \__keys_inherit:n #1
  {
    \__keys_undefine:
    \clist_set:Nn \l__keys_inherit_clist {#1}
    \cs_set_eq:cN { \c__keys_inherit_root_str \l_keys_path_str }
      \l__keys_inherit_clist
  }
```

(*End of definition for* `\__keys_inherit:n`.)

`\__keys_initialise:n`

A set up for initialisation: just run the code if it exists. We need to set the key string here, using the deprecated `tl var` as a piece of scratch space.

```
\cs_new_protected:Npn \__keys_initialise:n #1
  {
    \cs_if_exist:cTF
      { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
      { \__keys_execute_inherit: }
      {
        \str_clear:N \l__keys_inherit_str
        \cs_if_exist:cT { \c__keys_code_root_str \l_keys_path_str }
          {
            \exp_after:wN \__keys_find_key_module:wNN
              \l_keys_path_str \s__keys_stop
                \l_keys_key_tl \l_keys_key_str
            \tl_set_eq:NN \l_keys_key_tl \l_keys_key_str
            \tl_set:Nn \l_keys_value_tl {#1}
            \__keys_execute:no \l_keys_path_str \l_keys_value_tl
          }
      }
  }
```

(*End of definition for* `\__keys_initialise:n`.)

`\__keys_legacy_if_set:nn`
`\__keys_legacy_if_inverse:nn`
`\__keys_legacy_if_inverse:nnnn`

Much the same as expl3 booleans, except we assume that the switch exists.

```
\cs_new_protected:Npn \__keys_legacy_if_set:nn #1#2
  { \__keys_legacy_if_set:nnnn {#1} {#2} { true } { false } }
\cs_new_protected:Npn \__keys_legacy_if_set_inverse:nn #1#2
  { \__keys_legacy_if_set:nnnn {#1} {#2} { false } { true } }
\cs_new_protected:Npn \__keys_legacy_if_set:nnnn #1#2#3#4
  {
    \__keys_choice_make:
    \__keys_cmd_set:ne { \l_keys_path_str / true }
      { \exp_not:c { legacy_if_#2  set_ #3 :n } { \exp_not:n {#1} } }
    \__keys_cmd_set:ne { \l_keys_path_str / false }
      { \exp_not:c { legacy_if_#2  set_ #4 :n } { \exp_not:n {#1} } }
    \__keys_cmd_set:nn { \l_keys_path_str / unknown }
      {
        \msg_error:nne { keys } { boolean-values-only }
          \l_keys_path_str
      }
    \__keys_default_set:n { true }
    \cs_if_exist:cF { if#1 }
      {
        \cs:w newif \exp_after:wN \cs_end:
          \cs:w if#1 \cs_end:
      }
  }
```

(*End of definition for* `\__keys_legacy_if_set:nn`, `\__keys_legacy_if_inverse:nn`, *and* `\__keys_legacy_if_inverse:nnnn`.)

__keys_meta_make:n
__keys_meta_make:nn

To create a meta-key, simply set up to pass data through. The internal function is used here as a meta key should respect the prevailing filtering, etc.

```
\cs_new_protected:Npn \__keys_meta_make:n #1
  {
    \exp_args:NVo \__keys_cmd_set_direct:nn \l_keys_path_str
      {
        \exp_after:wN \__keys_set:nn \exp_after:wN
          { \l__keys_module_str } {#1}
      }
  }
\cs_new_protected:Npn \__keys_meta_make:nn #1#2
  {
    \exp_args:NV \__keys_cmd_set_direct:nn
      \l_keys_path_str { \__keys_set:nn {#1} {#2} }
  }
```

(*End of definition for* `\__keys_meta_make:n` *and* `\__keys_meta_make:nn`.)

__keys_prop_put:Nn
__keys_prop_put:cn

Much the same as other variables, but needs a dedicated auxiliary.

```
\cs_new_protected:Npn \__keys_prop_put:Nn #1#2
  {
    \prop_if_exist:NF #1 { \prop_new:N #1 }
    \exp_after:wN \__keys_find_key_module:wNN \l_keys_path_str \s__keys_stop
      \l__keys_tmpa_tl \l__keys_tmpb_tl
    \__keys_cmd_set:ne \l_keys_path_str
      {
        \exp_not:c { prop_ #2 put:Nnn }
        \exp_not:N #1
        { \l__keys_tmpb_tl }
        \exp_not:n { {##1} }
      }
  }
\cs_generate_variant:Nn \__keys_prop_put:Nn { c }
```

(*End of definition for* `\__keys_prop_put:Nn`.)

__keys_undefine:

Undefining a key has to be done without `\cs_undefine:c` as that function acts globally.

```
\cs_new_protected:Npn \__keys_undefine:
  {
    \clist_map_inline:nn
      { code , default , groups , inherit , type , check }
      {
        \__keys_cs_undefine:c
          { \tl_use:c { c__keys_ ##1 _root_str } \l_keys_path_str }
      }
  }
```

(*End of definition for* `\__keys_undefine:`.)

__keys_value_requirement:nn
__keys_check_forbidden:
__keys_check_required:

Validating key input is done using a second function which runs before the main key code. Setting that up means setting it equal to a generic stub which does the check. This approach makes the lookup very fast at the cost of one additional csname per key that needs it. The cleanup here has to know the structure of the following code.

```
\cs_new_protected:Npn \__keys_value_requirement:nn #1#2
```

```
  {
    \str_case:nnF {#2}
      {
        { true }
          {
            \cs_set_eq:cc
              { \c__keys_check_root_str \l_keys_path_str }
              { __keys_check_ #1 : }
          }
        { false }
          {
            \cs_if_eq:ccT
              { \c__keys_check_root_str \l_keys_path_str }
              { __keys_check_ #1 : }
              {
                \__keys_cs_undefine:c
                  { \c__keys_check_root_str \l_keys_path_str }
              }
          }
      }
      {
        \msg_error:nne { keys }
          { boolean-values-only }
          { .value_ #1 :n }
      }
  }
\cs_new_protected:Npn \__keys_check_forbidden:
  {
    \bool_if:NF \l__keys_no_value_bool
      {
        \msg_error:nnee { keys } { value-forbidden }
          \l_keys_path_str \l_keys_value_tl
        \use_none:nnn
      }
  }
\cs_new_protected:Npn \__keys_check_required:
  {
    \bool_if:NT \l__keys_no_value_bool
      {
        \msg_error:nne { keys } { value-required }
          \l_keys_path_str
        \use_none:nnn
      }
  }
```

(*End of definition for* `\__keys_value_requirement:nn`*,* `\__keys_check_forbidden:`*, and* `\__keys_check_required:`*.*)

__keys_usage:n
__keys_usage:NN
__keys_usage:w

Save the relevant data.

```
\cs_new_protected:Npn \__keys_usage:n #1
  {
    \str_case:nnF {#1}
      {
        { general }
```

```
          {
            \__keys_usage:NN \l_keys_usage_load_prop
              \c_false_bool
            \__keys_usage:NN \l_keys_usage_preamble_prop
              \c_false_bool
          }
        { load }
          {
            \__keys_usage:NN \l_keys_usage_load_prop
              \c_true_bool
            \__keys_usage:NN \l_keys_usage_preamble_prop
              \c_false_bool
          }
        { preamble }
          {
            \__keys_usage:NN \l_keys_usage_load_prop
              \c_false_bool
            \__keys_usage:NN \l_keys_usage_preamble_prop
              \c_true_bool
          }
      }
      {
        \msg_error:nnnn { keys }
          { choice-unknown }
          { .usage:n }
          {#1}
      }
  }
\cs_new_protected:Npn \__keys_usage:NN #1#2
  {
    \prop_get:NVNF #1 \l__keys_module_str \l__keys_tmpa_tl
      { \tl_clear:N \l__keys_tmpa_tl }
    \tl_set:Ne \l__keys_tmpb_tl
      { \exp_after:wN \__keys_usage:w \l_keys_path_str \s__keys_stop }
    \bool_if:NTF #2
      { \clist_put_right:NV \l__keys_tmpa_tl \l__keys_tmpb_tl }
      { \clist_remove_all:NV \l__keys_tmpa_tl \l__keys_tmpb_tl }
    \prop_put:NVV #1 \l__keys_module_str
      \l__keys_tmpa_tl
  }
\cs_new:Npn \__keys_usage:w #1 / #2 \s__keys_stop {#2}
```

(*End of definition for* `\__keys_usage:n`*,* `\__keys_usage:NN`*, and* `\__keys_usage:w`*.*)

`\__keys_variable_set:NnnN`
`\__keys_variable_set:cnnN`
`\__keys_variable_set_required:NnnN`
`\__keys_variable_set_required:cnnN`

Setting a variable takes the type and scope separately so that it is easy to make a new variable if needed.

```
\cs_new_protected:Npn \__keys_variable_set:NnnN #1#2#3#4
  {
    \use:c { #2_if_exist:NF } #1 { \use:c { #2 _new:N } #1 }
    \__keys_cmd_set:ne \l_keys_path_str
      {
        \exp_not:c { #2 _ #3 set:N #4 }
        \exp_not:N #1
        \exp_not:n  { {##1} }
```

```
      }
  }
\cs_generate_variant:Nn \__keys_variable_set:NnnN { c }
\cs_new_protected:Npn \__keys_variable_set_required:NnnN #1#2#3#4
  {
    \__keys_variable_set:NnnN #1 {#2} {#3} #4
    \__keys_value_requirement:nn { required } { true }
  }
\cs_generate_variant:Nn \__keys_variable_set_required:NnnN { c }
```

(*End of definition for* `\__keys_variable_set:NnnN` *and* `\__keys_variable_set_required:NnnN`*.*)

65.5 Creating key properties

The key property functions are all wrappers for internal functions, meaning that things stay readable and can also be altered later on.

Importantly, while key properties have "normal" argument specs, the underlying code always supplies one braced argument to these. As such, argument expansion is handled by hand rather than using the standard tools. This shows up particularly for the two-argument properties, where things would otherwise go badly wrong.

`.bool_set:N`
`.bool_set:c`
`.bool_gset:N`
`.bool_gset:c`

One function for this.

```
\cs_new_protected:cpn { \c__keys_props_root_str .bool_set:N } #1
  { \__keys_bool_set:Nn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .bool_set:c } #1
  { \__keys_bool_set:cn {#1} { } }
\cs_new_protected:cpn { \c__keys_props_root_str .bool_gset:N } #1
  { \__keys_bool_set:Nn #1 { g } }
\cs_new_protected:cpn { \c__keys_props_root_str .bool_gset:c } #1
  { \__keys_bool_set:cn {#1} { g } }
```

(*End of definition for* `.bool_set:N` *and* `.bool_gset:N`*. These functions are documented on page 240.*)

`.bool_set_inverse:N`
`.bool_set_inverse:c`
`.bool_gset_inverse:N`
`.bool_gset_inverse:c`

One function for this.

```
\cs_new_protected:cpn { \c__keys_props_root_str .bool_set_inverse:N } #1
  { \__keys_bool_set_inverse:Nn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .bool_set_inverse:c } #1
  { \__keys_bool_set_inverse:cn {#1} { } }
\cs_new_protected:cpn { \c__keys_props_root_str .bool_gset_inverse:N } #1
  { \__keys_bool_set_inverse:Nn #1 { g } }
\cs_new_protected:cpn { \c__keys_props_root_str .bool_gset_inverse:c } #1
  { \__keys_bool_set_inverse:cn {#1} { g } }
```

(*End of definition for* `.bool_set_inverse:N` *and* `.bool_gset_inverse:N`*. These functions are documented on page 240.*)

`.choice:`

Making a choice is handled internally, as it is also needed by `.generate_choices:n`.

```
\cs_new_protected:cpn { \c__keys_props_root_str .choice: }
  { \__keys_choice_make: }
```

(*End of definition for* `.choice:`*. This function is documented on page 240.*)

.choices:nn
.choices:Vn
.choices:en
.choices:on
.choices:xn

For auto-generation of a series of mutually-exclusive choices. Here, #1 consists of two separate arguments, hence the slightly odd-looking implementation.

```
\cs_new_protected:cpn { \c__keys_props_root_str .choices:nn } #1
  { \__keys_choices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_str .choices:Vn } #1
  { \exp_args:NV \__keys_choices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_str .choices:en } #1
  { \exp_args:Ne \__keys_choices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_str .choices:on } #1
  { \exp_args:No \__keys_choices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_str .choices:xn } #1
  { \exp_args:Nx \__keys_choices_make:nn #1 }
```

(*End of definition for* `.choices:nn`. *This function is documented on page 240.*)

.code:n

Creating code is simply a case of passing through to the underlying `set` function.

```
\cs_new_protected:cpn { \c__keys_props_root_str .code:n } #1
  { \__keys_cmd_set:nn \l_keys_path_str {#1} }
```

(*End of definition for* `.code:n`. *This function is documented on page 241.*)

.clist_set:N
.clist_set:c
.clist_gset:N
.clist_gset:c

```
\cs_new_protected:cpn { \c__keys_props_root_str .clist_set:N } #1
  { \__keys_variable_set:NnnN #1 { clist } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .clist_set:c } #1
  { \__keys_variable_set:cnnN {#1} { clist } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .clist_gset:N } #1
  { \__keys_variable_set:NnnN #1 { clist } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .clist_gset:c } #1
  { \__keys_variable_set:cnnN {#1} { clist } { g } n }
```

(*End of definition for* `.clist_set:N` *and* `.clist_gset:N`. *These functions are documented on page 240.*)

.cs_set:Np
.cs_set:cp
.cs_set_protected:Np
.cs_set_protected:cp
.cs_gset:Np
.cs_gset:cp
.cs_gset_protected:Np
.cs_gset_protected:cp

```
\cs_new_protected:cpn { \c__keys_props_root_str .cs_set:Np } #1
  { \__keys_cs_set:NNpn \cs_set:Npn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .cs_set:cp } #1
  { \__keys_cs_set:Ncpn \cs_set:Npn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .cs_set_protected:Np } #1
  { \__keys_cs_set:NNpn \cs_set_protected:Npn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .cs_set_protected:cp } #1
  { \__keys_cs_set:Ncpn \cs_set_protected:Npn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .cs_gset:Np } #1
  { \__keys_cs_set:NNpn \cs_gset:Npn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .cs_gset:cp } #1
  { \__keys_cs_set:Ncpn \cs_gset:Npn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .cs_gset_protected:Np } #1
  { \__keys_cs_set:NNpn \cs_gset_protected:Npn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .cs_gset_protected:cp } #1
  { \__keys_cs_set:Ncpn \cs_gset_protected:Npn #1 { } }
```

(*End of definition for* `.cs_set:Np` *and others. These functions are documented on page 241.*)

.default:n
.default:V
.default:e
.default:o
.default:x

Expansion is left to the internal functions.

```
\cs_new_protected:cpn { \c__keys_props_root_str .default:n } #1
  { \__keys_default_set:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_str .default:V } #1
  { \exp_args:NV \__keys_default_set:n #1 }
\cs_new_protected:cpn { \c__keys_props_root_str .default:e } #1
  { \exp_args:Ne \__keys_default_set:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_str .default:o } #1
  { \exp_args:No \__keys_default_set:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_str .default:x } #1
  { \exp_args:Nx \__keys_default_set:n {#1} }
```

(*End of definition for `.default:n`. This function is documented on page 241.*)

.dim_set:N
.dim_set:c
.dim_gset:N
.dim_gset:c

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_str .dim_set:N } #1
  { \__keys_variable_set_required:NnnN #1 { dim } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .dim_set:c } #1
  { \__keys_variable_set_required:cnnN {#1} { dim } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .dim_gset:N } #1
  { \__keys_variable_set_required:NnnN #1 { dim } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .dim_gset:c } #1
  { \__keys_variable_set_required:cnnN {#1} { dim } { g } n }
```

(*End of definition for `.dim_set:N` and `.dim_gset:N`. These functions are documented on page 241.*)

.fp_set:N
.fp_set:c
.fp_gset:N
.fp_gset:c

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_str .fp_set:N } #1
  { \__keys_variable_set_required:NnnN #1 { fp } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .fp_set:c } #1
  { \__keys_variable_set_required:cnnN {#1} { fp } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .fp_gset:N } #1
  { \__keys_variable_set_required:NnnN #1 { fp } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .fp_gset:c } #1
  { \__keys_variable_set_required:cnnN {#1} { fp } { g } n }
```

(*End of definition for `.fp_set:N` and `.fp_gset:N`. These functions are documented on page 241.*)

.groups:n

A single property to create groups of keys.

```
\cs_new_protected:cpn { \c__keys_props_root_str .groups:n } #1
  { \__keys_groups_set:n {#1} }
```

(*End of definition for `.groups:n`. This function is documented on page 242.*)

.inherit:n

Nothing complex: only one variant at the moment!

```
\cs_new_protected:cpn { \c__keys_props_root_str .inherit:n } #1
  { \__keys_inherit:n {#1} }
```

(*End of definition for `.inherit:n`. This function is documented on page 242.*)

.initial:n
.initial:V
.initial:e
.initial:o
.initial:x

The standard hand-off approach.

```
\cs_new_protected:cpn { \c__keys_props_root_str .initial:n } #1
  { \__keys_initialise:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_str .initial:V } #1
  { \exp_args:NV \__keys_initialise:n #1 }
\cs_new_protected:cpn { \c__keys_props_root_str .initial:e } #1
  { \exp_args:Ne \__keys_initialise:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_str .initial:o } #1
  { \exp_args:No \__keys_initialise:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_str .initial:x } #1
  { \exp_args:Nx \__keys_initialise:n {#1} }
```

(*End of definition for* `.initial:n`. *This function is documented on page 242.*)

.int_set:N
.int_set:c
.int_gset:N
.int_gset:c

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_str .int_set:N } #1
  { \__keys_variable_set_required:NnnN #1 { int } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .int_set:c } #1
  { \__keys_variable_set_required:cnnN {#1} { int } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .int_gset:N } #1
  { \__keys_variable_set_required:NnnN #1 { int } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .int_gset:c } #1
  { \__keys_variable_set_required:cnnN {#1} { int } { g } n }
```

(*End of definition for* `.int_set:N` *and* `.int_gset:N`. *These functions are documented on page 242.*)

.legacy_if_set:n
.legacy_if_gset:n
.legacy_if_set_inverse:n
.legacy_if_gset_inverse:n

```
\cs_new_protected:cpn { \c__keys_props_root_str .legacy_if_set:n } #1
  { \__keys_legacy_if_set:nn {#1} { } }
\cs_new_protected:cpn { \c__keys_props_root_str .legacy_if_gset:n } #1
  { \__keys_legacy_if_set:nn {#1} { g } }
\cs_new_protected:cpn { \c__keys_props_root_str .legacy_if_set_inverse:n } #1
  { \__keys_legacy_if_set_inverse:nn {#1} { } }
\cs_new_protected:cpn { \c__keys_props_root_str .legacy_if_gset_inverse:n } #1
  { \__keys_legacy_if_set_inverse:nn {#1} { g } }
```

(*End of definition for* `.legacy_if_set:n` *and others. These functions are documented on page 242.*)

.meta:n

Making a meta is handled internally.

```
\cs_new_protected:cpn { \c__keys_props_root_str .meta:n } #1
  { \__keys_meta_make:n {#1} }
```

(*End of definition for* `.meta:n`. *This function is documented on page 242.*)

.meta:nn

Meta with path: potentially lots of variants, but for the moment no so many defined.

```
\cs_new_protected:cpn { \c__keys_props_root_str .meta:nn } #1
  { \__keys_meta_make:nn #1 }
```

(*End of definition for* `.meta:nn`. *This function is documented on page 243.*)

.multichoice:
.multichoices:nn
.multichoices:Vn
.multichoices:en
.multichoices:on
.multichoices:xn

The same idea as `.choice:` and `.choices:nn`, but where more than one choice is allowed.

```
\cs_new_protected:cpn { \c__keys_props_root_str .multichoice: }
  { \__keys_multichoice_make: }
\cs_new_protected:cpn { \c__keys_props_root_str .multichoices:nn } #1
  { \__keys_multichoices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_str .multichoices:Vn } #1
  { \exp_args:NV \__keys_multichoices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_str .multichoices:en } #1
  { \exp_args:Ne \__keys_multichoices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_str .multichoices:on } #1
  { \exp_args:No \__keys_multichoices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_str .multichoices:xn } #1
  { \exp_args:Nx \__keys_multichoices_make:nn #1 }
```

(*End of definition for* `.multichoice:` *and* `.multichoices:nn`. *These functions are documented on page* *243*.)

.muskip_set:N
.muskip_set:c
.muskip_gset:N
.muskip_gset:c

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_str .muskip_set:N } #1
  { \__keys_variable_set_required:NnnN #1 { muskip } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .muskip_set:c } #1
  { \__keys_variable_set_required:cnnN {#1} { muskip } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .muskip_gset:N } #1
  { \__keys_variable_set_required:NnnN #1 { muskip } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .muskip_gset:c } #1
  { \__keys_variable_set_required:cnnN {#1} { muskip } { g } n }
```

(*End of definition for* `.muskip_set:N` *and* `.muskip_gset:N`. *These functions are documented on page* *243*.)

.prop_put:N
.prop_put:c
.prop_gput:N
.prop_gput:c

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_str .prop_put:N } #1
  { \__keys_prop_put:Nn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_str .prop_put:c } #1
  { \__keys_prop_put:cn {#1} { } }
\cs_new_protected:cpn { \c__keys_props_root_str .prop_gput:N } #1
  { \__keys_prop_put:Nn #1 { g } }
\cs_new_protected:cpn { \c__keys_props_root_str .prop_gput:c } #1
  { \__keys_prop_put:cn {#1} { g } }
```

(*End of definition for* `.prop_put:N` *and* `.prop_gput:N`. *These functions are documented on page* *243*.)

.skip_set:N
.skip_set:c
.skip_gset:N
.skip_gset:c

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_str .skip_set:N } #1
  { \__keys_variable_set_required:NnnN #1 { skip } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .skip_set:c } #1
  { \__keys_variable_set_required:cnnN {#1} { skip } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .skip_gset:N } #1
  { \__keys_variable_set_required:NnnN #1 { skip } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .skip_gset:c } #1
  { \__keys_variable_set_required:cnnN {#1} { skip } { g } n }
```

(*End of definition for* `.skip_set:N` *and* `.skip_gset:N`. *These functions are documented on page* *243*.)

.str_set:N
.str_set:c
.str_gset:N
.str_gset:c
.str_set_e:N
.str_set_e:c
.str_gset_e:N
.str_gset_e:c

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_str .str_set:N } #1
  { \__keys_variable_set:NnnN #1 { str } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .str_set:c } #1
  { \__keys_variable_set:cnnN {#1} { str } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .str_set_e:N } #1
  { \__keys_variable_set:NnnN #1 { str } { } e }
\cs_new_protected:cpn { \c__keys_props_root_str .str_set_e:c } #1
  { \__keys_variable_set:cnnN {#1} { str } { } e }
\cs_new_protected:cpn { \c__keys_props_root_str .str_gset:N } #1
  { \__keys_variable_set:NnnN #1 { str } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .str_gset:c } #1
  { \__keys_variable_set:cnnN {#1} { str } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .str_gset_e:N } #1
  { \__keys_variable_set:NnnN #1 { str } { g } e }
\cs_new_protected:cpn { \c__keys_props_root_str .str_gset_e:c } #1
  { \__keys_variable_set:cnnN {#1} { str } { g } e }
```

(*End of definition for* `.str_set:N` *and others. These functions are documented on page 243.*)

.tl_set:N
.tl_set:c
.tl_gset:N
.tl_gset:c
.tl_set_e:N
.tl_set_e:c
.tl_gset_e:N
.tl_gset_e:c

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_str .tl_set:N } #1
  { \__keys_variable_set:NnnN #1 { tl } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_set:c } #1
  { \__keys_variable_set:cnnN {#1} { tl } { } n }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_set_e:N } #1
  { \__keys_variable_set:NnnN #1 { tl } { } e }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_set_e:c } #1
  { \__keys_variable_set:cnnN {#1} { tl } { } e }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_gset:N } #1
  { \__keys_variable_set:NnnN #1 { tl } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_gset:c } #1
  { \__keys_variable_set:cnnN {#1} { tl } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_e:N } #1
  { \__keys_variable_set:NnnN #1 { tl } { g } e }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_e:c } #1
  { \__keys_variable_set:cnnN {#1} { tl } { g } e }
```

(*End of definition for* `.tl_set:N` *and others. These functions are documented on page 244.*)

.undefine:

Another simple wrapper.

```
\cs_new_protected:cpn { \c__keys_props_root_str .undefine: }
  { \__keys_undefine: }
```

(*End of definition for* `.undefine:`*. This function is documented on page 244.*)

.usage:n

```
\cs_new_protected:cpn { \c__keys_props_root_str .usage:n } #1
  { \__keys_usage:n {#1} }
```

(*End of definition for* `.usage:n`*. This function is documented on page 247.*)

.value_forbidden:n
.value_required:n

These are very similar, so both call the same function.

```
\cs_new_protected:cpn { \c__keys_props_root_str .value_forbidden:n } #1
  { \__keys_value_requirement:nn { forbidden } {#1} }
\cs_new_protected:cpn { \c__keys_props_root_str .value_required:n } #1
  { \__keys_value_requirement:nn { required } {#1} }
```

(*End of definition for* `.value_forbidden:n` *and* `.value_required:n`*. These functions are documented on page* *244*.)

65.6 Setting keys

__keys_set:nnnnNn
__keys_set:nnnnnnnNn
__keys_reset_bool:N
__keys_reset_var:N
__keys_set:nn
__keys_set:nnn

The aim here is to allow nesting of key setting without needing lots of tracking. That is done by expanding the appropriate tokens "around" the core keyval parsing. As there are several different sub-paths, this needs a few steps and some generic auxiliaries. The arguments here are

1. The root for keys
2. The key groups
3. The keys themselves
4. The relative root for return of unset keys
5. The `clist` var for returning unset keys
6. The code to set up the correct selection approach

```
\cs_new_protected:Npn \__keys_set:nnnnNn
  {
    \exp_args:Nooo \__keys_set:nnnnnnnNn
      \l__keys_unused_clist
      \l__keys_selective_clist
      \l__keys_relative_tl
  }
\cs_new_protected:Npn \__keys_set:nnnnnnnNn #1#2#3#4#5#6#7#8#9
  {
    \clist_clear:N \l__keys_unused_clist
    \clist_set:Ne \l__keys_selective_clist { \tl_to_str:n {#5} }
    \tl_set:Nn \l__keys_relative_tl {#7}
    \use:e
      {
        \exp_not:n
          {
            #9
            \__keys_set:nn {#4} {#6}
          }
        \__keys_reset_bool:N \l__keys_only_known_bool
        \__keys_reset_bool:N \l__keys_exclude_bool
        \__keys_reset_bool:N \l__keys_selective_bool
      }
    \clist_set_eq:NN #8 \l__keys_unused_clist
    \__kernel_tl_set:Nx \l__keys_unused_clist { \exp_not:n {#1} }
    \__kernel_tl_set:Nx \l__keys_selective_clist {#2}
    \__kernel_tl_set:Nx \l__keys_relative_tl { \exp_not:n {#3} }
```

```
  }
\cs_new:Npn \__keys_reset_bool:N #1
  {
    \exp_not:c
      { bool_set_ \bool_if:NTF #1 { true } { false } :N }
        \exp_not:N #1
  }
\cs_new_protected:Npn \__keys_set:nn #1#2
  { \exp_args:No \__keys_set:nnn \l__keys_module_str {#1} {#2} }
\cs_new_protected:Npn \__keys_set:nnn #1#2#3
  {
    \str_set:Ne \l__keys_module_str { \__keys_trim_spaces:n {#2} }
    \keyval_parse:NNn \__keys_set_keyval:n \__keys_set_keyval:nn {#3}
    \str_set:Nn \l__keys_module_str {#1}
  }
```

(*End of definition for* `\__keys_set:nnnnNn` *and others.*)

`\keys_set:nn`
`\keys_set:nV`
`\keys_set:nv`
`\keys_set:ne`
`\keys_set:no`
`\keys_set:nx`

A simple wrapper allowing for nesting.

```
\cs_new_protected:Npn \keys_set:nn #1#2
  {
    \__keys_set:nnnnNn
      {#1} { } {#2} { \q__keys_no_value } \l__keys_tmp_clist
      {
        \bool_set_false:N \l__keys_only_known_bool
        \bool_set_false:N \l__keys_exclude_bool
        \bool_set_false:N \l__keys_selective_bool
      }
  }
\cs_generate_variant:Nn \keys_set:nn { nV , nv , ne , no , nx }
```

(*End of definition for* `\keys_set:nn`*. This function is documented on page 247.*)

`\keys_set_known:nnnN`
`\keys_set_known:nVnN`
`\keys_set_known:nvnN`
`\keys_set_known:nenN`
`\keys_set_known:nonN`
`\keys_set_known:nnN`
`\keys_set_known:nVN`
`\keys_set_known:nvN`
`\keys_set_known:neN`
`\keys_set_known:noN`
`\keys_set_known:nn`
`\keys_set_known:nV`
`\keys_set_known:nv`
`\keys_set_known:ne`
`\keys_set_known:no`

Simply set the right variables.

```
\cs_new_protected:Npn \keys_set_known:nnnN #1#2#3#4
  {
    \__keys_set:nnnnNn
      {#1} { } {#2} {#3} #4
      {
        \bool_set_true:N \l__keys_only_known_bool
        \bool_set_false:N \l__keys_exclude_bool
        \bool_set_false:N \l__keys_selective_bool
      }
  }
\cs_generate_variant:Nn \keys_set_known:nnnN { nV , nv , ne , no }
\cs_new_protected:Npn \keys_set_known:nnN #1#2#3
  { \keys_set_known:nnnN {#1} {#2} { \q__keys_no_value } #3 }
\cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , ne , no }
\cs_new_protected:Npn \keys_set_known:nn #1#2
  { \keys_set_known:nnnN {#1} {#2} { \q__keys_no_value } \l__keys_tmp_clist }
\cs_generate_variant:Nn \keys_set_known:nn { nV , nv , ne , no }
```

(*End of definition for* `\keys_set_known:nnnN`*,* `\keys_set_known:nnN`*, and* `\keys_set_known:nn`*. These functions are documented on page 249.*)

\keys_set_exclude_groups:nnnN
\keys_set_exclude_groups:nnVN
\keys_set_exclude_groups:nnvN
\keys_set_exclude_groups:nnoN
\keys_set_exclude_groups:nnnnN
\keys_set_exclude_groups:nnVnN
\keys_set_exclude_groups:nnvnN
\keys_set_exclude_groups:nnonN
\keys_set_exclude_groups:nnn
\keys_set_exclude_groups:nnV
\keys_set_exclude_groups:nnv
\keys_set_exclude_groups:nno
\keys_set_groups:nnnN
\keys_set_groups:nnVN
\keys_set_groups:nnvN
\keys_set_groups:nnoN
\keys_set_groups:nnnnN
\keys_set_groups:nnVnN
\keys_set_groups:nnvnN
\keys_set_groups:nnonN
\keys_set_groups:nnn
\keys_set_groups:nnV
\keys_set_groups:nnv
\keys_set_groups:nno

The same for (exclusion) groups.

```
\cs_new_protected:Npn \keys_set_exclude_groups:nnnnN #1#2#3#4#5
  {
    \__keys_set:nnnnNn
      {#1} {#2} {#3} {#4} #5
      {
        \bool_set_false:N \l__keys_only_known_bool
        \bool_set_true:N \l__keys_exclude_bool
        \bool_set_true:N \l__keys_selective_bool
      }
  }
\cs_generate_variant:Nn \keys_set_exclude_groups:nnnnN { nnV , nnv , nno }
\cs_new_protected:Npn \keys_set_exclude_groups:nnnN #1#2#3#4
  { \keys_set_exclude_groups:nnnnN {#1} {#2} {#3} { \q__keys_no_value } #4 }
\cs_generate_variant:Nn \keys_set_exclude_groups:nnnN { nnV , nnv , nno }
\cs_new_protected:Npn \keys_set_exclude_groups:nnn #1#2#3
  {
    \keys_set_exclude_groups:nnnnN {#1} {#2} {#3}
      { \q__keys_no_value } \l__keys_tmp_clist
  }
\cs_generate_variant:Nn \keys_set_exclude_groups:nnn { nnV , nnv , nno }
\cs_new_protected:Npn \keys_set_groups:nnnnN #1#2#3#4#5
  {
    \__keys_set:nnnnNn
      {#1} {#2} {#3} {#4} #5
      {
        \bool_set_false:N \l__keys_only_known_bool
        \bool_set_false:N \l__keys_exclude_bool
        \bool_set_true:N \l__keys_selective_bool
      }
  }
\cs_generate_variant:Nn \keys_set_groups:nnnnN { nnV , nnv , nno }
\cs_new_protected:Npn \keys_set_groups:nnnN #1#2#3#4
  { \keys_set_groups:nnnnN {#1} {#2} {#3} { \q__keys_no_value } #4 }
\cs_generate_variant:Nn \keys_set_groups:nnnN { nnV , nnv , nno }
\cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
  {
    \keys_set_groups:nnnnN {#1} {#2} {#3}
      { \q__keys_no_value } \l__keys_tmp_clist
  }
\cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }
```

(*End of definition for* \keys_set_exclude_groups:nnnN *and others. These functions are documented on page 250.*)

\keys_precompile:nnN

A simple wrapper.

```
\cs_new_protected:Npn \keys_precompile:nnN #1#2#3
  {
    \bool_set_true:N \l__keys_precompile_bool
    \tl_clear:N \l__keys_precompile_tl
    \keys_set:nn {#1} {#2}
    \bool_set_false:N \l__keys_precompile_bool
    \tl_set_eq:NN #3 \l__keys_precompile_tl
  }
```

(*End of definition for* `\keys_precompile:nnN`. *This function is documented on page* 250.)

__keys_set_keyval:n
__keys_set_keyval:nn
__keys_set_keyval:nnn
__keys_set_keyval:onn
__keys_find_key_module:wNN
__keys_find_key_module_auxi:Nw
__keys_find_key_module_auxii:Nw
__keys_find_key_module_auxiii:Nn
__keys_find_key_module_auxiv:Nw
__keys_set_selective:

A shared system once again. First, set the current path and add a default if needed. There are then checks to see if a value is required or forbidden. If everything passes, move on to execute the code.

```
\cs_new_protected:Npn \__keys_set_keyval:n #1
  {
    \bool_set_true:N \l__keys_no_value_bool
    \__keys_set_keyval:onn \l__keys_module_str {#1} { }
  }
\cs_new_protected:Npn \__keys_set_keyval:nn #1#2
  {
    \bool_set_false:N \l__keys_no_value_bool
    \__keys_set_keyval:onn \l__keys_module_str {#1} {#2}
  }
```

The key path here can be fully defined, after which there is a search for the key and module names: the user may have passed them with part of what is actually the module (for our purposes) in the key name. As that happens on a per-key basis, we use the stack approach to restore the module name without a group.

```
\cs_new_protected:Npn \__keys_set_keyval:nnn #1#2#3
  {
    \__kernel_tl_set:Nx \l_keys_path_str
      {
        \tl_if_blank:nF {#1}
          { #1 / }
        \__keys_trim_spaces:n {#2}
      }
    \str_clear:N \l__keys_module_str
    \str_clear:N \l__keys_inherit_str
    \exp_after:wN \__keys_find_key_module:wNN \l_keys_path_str \s__keys_stop
      \l__keys_module_str \l_keys_key_str
    \tl_set_eq:NN \l_keys_key_tl \l_keys_key_str
    \__keys_value_or_default:n {#3}
    \bool_if:NTF \l__keys_selective_bool
      \__keys_set_selective:
      \__keys_execute:
    \str_set:Nn \l__keys_module_str {#1}
  }
\cs_generate_variant:Nn \__keys_set_keyval:nnn { o }
```

This function uses `\cs_set_nopar:Npe` internally for performance reasons, the argument `#1` is already a string in every usage, so turning it into a string again seems unnecessary.

```
\cs_new_protected:Npn \__keys_find_key_module:wNN #1 \s__keys_stop #2 #3
  {
    \__keys_find_key_module_auxi:Nw #2 #1 \s__keys_nil \__keys_find_key_module_auxii:Nw
      / \s__keys_nil \__keys_find_key_module_auxiv:Nw #3
  }
\cs_new_protected:Npn \__keys_find_key_module_auxi:Nw #1 #2 / #3 \s__keys_nil #4
  {
    #4 #1 #2 \s__keys_mark #3 \s__keys_nil #4
  }
\cs_new_protected:Npn \__keys_find_key_module_auxii:Nw
    #1 #2 \s__keys_mark #3 \s__keys_nil \__keys_find_key_module_auxii:Nw
```

```
  {
    \cs_set_nopar:Npe #1 { \tl_if_empty:NF #1 { #1 / } #2 }
    \__keys_find_key_module_auxi:Nw #1 #3 \s__keys_nil \__keys_find_key_module_auxiii:Nw
  }
\cs_new_protected:Npn \__keys_find_key_module_auxiii:Nw #1 #2 \s__keys_mark
  {
    \cs_set_nopar:Npe #1 { \tl_if_empty:NF #1 { #1 / } #2 }
    \__keys_find_key_module_auxi:Nw #1
  }
\cs_new_protected:Npn \__keys_find_key_module_auxiv:Nw
    #1 #2 \s__keys_nil #3 \s__keys_mark
    \s__keys_nil \__keys_find_key_module_auxiv:Nw #4
  {
    \cs_set_nopar:Npn #4 { #2 }
  }
```

If selective setting is active, there are a number of possible sub-cases to consider. The key name may not be known at all or if it is, it may not have any groups assigned. There is then the question of whether the selection is opt-in or opt-out.

```
\cs_new_protected:Npn \__keys_set_selective:
  {
    \cs_if_exist:cTF { \c__keys_groups_root_str \l_keys_path_str }
      {
        \clist_set_eq:Nc \l__keys_groups_clist
          { \c__keys_groups_root_str \l_keys_path_str }
        \__keys_check_groups:
      }
      {
        \bool_if:NTF \l__keys_exclude_bool
          \__keys_execute:
          \__keys_store_unused:
      }
  }
```

In the case where selective setting requires a comparison of the list of groups which apply to a key with the list of those which have been set active. That requires two mappings, and again a different outcome depending on whether opt-in or opt-out is set. It is safe to use `\clist_if_in:NnTF` because both `\l__keys_selective_clist` and `\l__keys_-groups_clist` contain the groups as strings, without leading/trailing spaces in any item, since the l3clist functions were applied to the result of applying `\tl_to_str:n`.

```
\cs_new_protected:Npn \__keys_check_groups:
  {
    \bool_set_false:N \l__keys_tmp_bool
    \clist_map_inline:Nn \l__keys_selective_clist
      {
        \clist_if_in:NnT \l__keys_groups_clist {##1}
          {
            \bool_set_true:N \l__keys_tmp_bool
            \clist_map_break:
          }
      }
    \bool_if:NTF \l__keys_tmp_bool
      {
        \bool_if:NTF \l__keys_exclude_bool
```

```
        \__keys_store_unused:
        \__keys_execute:
      }
      {
        \bool_if:NTF \l__keys_exclude_bool
          \__keys_execute:
          \__keys_store_unused:
      }
  }
```

(*End of definition for* `\__keys_set_keyval:n` *and others.*)

`\__keys_value_or_default:n` `\__keys_default_inherit:`

If a value is given, return it as `#1`, otherwise send a default if available.

```
\cs_new_protected:Npn \__keys_value_or_default:n #1
  {
    \bool_if:NTF \l__keys_no_value_bool
      {
        \cs_if_exist:cTF { \c__keys_default_root_str \l_keys_path_str }
          {
            \tl_set_eq:Nc
              \l_keys_value_tl
              { \c__keys_default_root_str \l_keys_path_str }
          }
          {
            \tl_clear:N \l_keys_value_tl
            \cs_if_exist:cT
              { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
              { \__keys_default_inherit: }
          }
      }
      { \tl_set:Nn \l_keys_value_tl {#1} }
  }
\cs_new_protected:Npn \__keys_default_inherit:
  {
    \clist_map_inline:cn
      { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
      {
        \cs_if_exist:cT
          { \c__keys_default_root_str ##1 / \l_keys_key_str }
          {
            \tl_set_eq:Nc
              \l_keys_value_tl
              { \c__keys_default_root_str ##1 / \l_keys_key_str }
            \clist_map_break:
          }
      }
  }
```

(*End of definition for* `\__keys_value_or_default:n` *and* `\__keys_default_inherit:`.)

`\__keys_execute:` `\__keys_execute_inherit:` `\__keys_execute_unknown:` `\__keys_execute:nn` `\__keys_execute:no` `\__keys_store_unused:` `\__keys_store_unused_aux:`

Actually executing a key is done in two parts. First, look for the key itself, then look for the `unknown` key with the same path. If both of these fail, complain. What exactly happens if a key is unknown depends on whether unknown keys are being skipped or if an error should be raised.

```
\cs_new_protected:Npn \__keys_execute:
  {
    \cs_if_exist:cTF { \c__keys_code_root_str \l_keys_path_str }
      {
        \cs_if_exist_use:c { \c__keys_check_root_str \l_keys_path_str }
        \__keys_execute:no \l_keys_path_str \l_keys_value_tl
      }
      {
        \cs_if_exist:cTF
          { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
          { \__keys_execute_inherit: }
          { \__keys_execute_unknown: }
      }
  }
```

To deal with the case where there is no hit, we leave `\__keys_execute_unknown:` in the input stream and clean it up using the break function: that avoids needing a boolean.

```
\cs_new_protected:Npn \__keys_execute_inherit:
  {
    \clist_map_inline:cn
      { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
      {
        \cs_if_exist:cT
          { \c__keys_code_root_str ##1 / \l_keys_key_str }
          {
            \str_set:Nn \l__keys_inherit_str {##1}
            \cs_if_exist_use:c { \c__keys_check_root_str ##1 / \l_keys_key_str }
            \__keys_execute:no { ##1 / \l_keys_key_str } \l_keys_value_tl
            \clist_map_break:n \use_none:n
          }
      }
    \__keys_execute_unknown:
  }
\cs_new_protected:Npn \__keys_execute_unknown:
  {
    \bool_if:NTF \l__keys_only_known_bool
      { \__keys_store_unused: }
      {
        \cs_if_exist:cTF
          { \c__keys_code_root_str \l__keys_module_str / unknown }
          {
            \bool_if:NT \l__keys_no_value_bool
              {
                \cs_if_exist:cT
                  { \c__keys_default_root_str \l__keys_module_str / unknown }
                  {
                    \tl_set_eq:Nc
                      \l_keys_value_tl
                      { \c__keys_default_root_str \l__keys_module_str / unknown }
                  }
              }
            \__keys_execute:no { \l__keys_module_str / unknown } \l_keys_value_tl
          }
          {
```

```
            \msg_error:nnee { keys } { unknown }
              \l_keys_path_str \l__keys_module_str
          }
      }
  }
```

A key's code is in the control sequence with csname \c__keys_code_root_str #1. We expand it once to get the replacement text (with argument #2) and call \use:n with this replacement as its argument. This ensures that any undefined control sequence error in the key's code will lead to an error message of the form <argument>...⟨*control sequence*⟩ in which one can read the (undefined) ⟨*control sequence*⟩ in full, rather than an error message that starts with the potentially very long key name, which would make the (undefined) ⟨*control sequence*⟩ be truncated or sometimes completely hidden. See https://github.com/latex3/latex2e/issues/351.

```
\cs_new:Npn \__keys_execute:nn #1#2
  { \__keys_execute:no {#1} { \prg_do_nothing: #2 } }
\cs_new:Npn \__keys_execute:no #1#2
  {
    \exp_args:NNo \exp_args:No \use:n
      {
        \cs:w \c__keys_code_root_str #1 \exp_after:wN \cs_end:
        \exp_after:wN {#2}
      }
  }
```

When there is no relative path, things here are easy: just save the key name and value. When we are working with a relative path, first we need to turn it into a string: that can't happen earlier as we need to store \q__keys_no_value. Then, use a standard delimited approach to fish out the partial path.

```
\cs_new_protected:Npn \__keys_store_unused:
  {
    \__keys_quark_if_no_value:NTF \l__keys_relative_tl
      {
        \clist_put_right:Ne \l__keys_unused_clist
          {
            \l_keys_key_str
            \bool_if:NF \l__keys_no_value_bool
              { = { \exp_not:o \l_keys_value_tl } }
          }
      }
      {
        \tl_if_empty:NTF \l__keys_relative_tl
          {
            \clist_put_right:Ne \l__keys_unused_clist
              {
                \l_keys_path_str
                \bool_if:NF \l__keys_no_value_bool
                  { = { \exp_not:o \l_keys_value_tl } }
              }
          }
          { \__keys_store_unused_aux: }
      }
  }
\cs_new_protected:Npn \__keys_store_unused_aux:
```

```
  {
    \__kernel_tl_set:Nx \l__keys_relative_tl
      { \exp_args:No \__keys_trim_spaces:n \l__keys_relative_tl }
    \use:e
      {
        \cs_set_protected:Npn \__keys_store_unused:w
          ##1 \l__keys_relative_tl /
          ##2 \l__keys_relative_tl /
          ##3 \s__keys_stop
      }
        {
          \tl_if_blank:nF {##1}
            {
              \msg_error:nnee { keys } { bad-relative-key-path }
                \l_keys_path_str
                \l__keys_relative_tl
            }
          \clist_put_right:Ne \l__keys_unused_clist
            {
              \exp_not:n {##2}
              \bool_if:NF \l__keys_no_value_bool
                { = { \exp_not:o \l_keys_value_tl } }
            }
        }
    \use:e
      {
        \__keys_store_unused:w \l_keys_path_str
          \l__keys_relative_tl / \l__keys_relative_tl /
          \s__keys_stop
      }
  }
\cs_new_protected:Npn \__keys_store_unused:w { }
```

(*End of definition for* `\__keys_execute:` *and others.*)

`\__keys_choice_find:n`
`\__keys_choice_find:nn`
`\__keys_multichoice_find:n`

Executing a choice has two parts. First, try the choice given, then if that fails call the unknown key. That always exists, as it is created when a choice is first made. So there is no need for any escape code. For multiple choices, the same code ends up used in a mapping.

```
\cs_new:Npn \__keys_choice_find:n #1
  {
    \str_if_empty:NTF \l__keys_inherit_str
      { \__keys_choice_find:nn \l_keys_path_str {#1} }
      {
        \__keys_choice_find:nn
          { \l__keys_inherit_str / \l_keys_key_str } {#1}
      }
  }
\cs_new:Npn \__keys_choice_find:nn #1#2
  {
    \cs_if_exist:cTF { \c__keys_code_root_str #1 / \__keys_trim_spaces:n {#2} }
      { \__keys_execute:nn { #1 / \__keys_trim_spaces:n {#2} } {#2} }
      { \__keys_execute:nn { #1 / unknown } {#2} }
  }
```

```
\cs_new:Npn \__keys_multichoice_find:n #1
  { \clist_map_function:nN {#1} \__keys_choice_find:n }
```

(*End of definition for* `\__keys_choice_find:n`*,* `\__keys_choice_find:nn`*, and* `\__keys_multichoice_find:n`*.*)

65.7 Utilities

`\__keys_parent:o`
`\__keys_parent_auxi:w`
`\__keys_parent_auxii:w`
`\__keys_parent_auxiii:n`
`\__keys_parent_auxiv:w`

Used to strip off the ending part of the key path after the last `/`.

```
\cs_new:Npn \__keys_parent:o #1
  {
    \exp_after:wN \__keys_parent_auxi:w #1 \q_nil \__keys_parent_auxii:w
      / \q_nil \__keys_parent_auxiv:w
  }
\cs_new:Npn \__keys_parent_auxi:w #1 / #2 \q_nil #3
  {
    #3 { #1 } #2 \q_nil #3
  }
\cs_new:Npn \__keys_parent_auxii:w #1 #2 \q_nil \__keys_parent_auxii:w
  {
    #1 \__keys_parent_auxi:w #2 \q_nil \__keys_parent_auxiii:n
  }
\cs_new:Npn \__keys_parent_auxiii:n #1
  {
    / #1 \__keys_parent_auxi:w
  }
\cs_new:Npn \__keys_parent_auxiv:w #1 \q_nil \__keys_parent_auxiv:w
  {
  }
```

(*End of definition for* `\__keys_parent:o` *and others.*)

`\__keys_trim_spaces:n`
`\__keys_trim_spaces_auxi:w`
`\__keys_trim_spaces_auxii:w`
`\__keys_trim_spaces_auxiii:w`

Space stripping has to allow for the fact that the key here might have several parts, and spaces need to be stripped from each part. Since the key name is turned into a string groups can't be stripped accidentally and the precautions of `\tl_trim_spaces:n` aren't necessary, in this case it is much faster to just directly strip spaces around `/`.

```
\group_begin:
  \cs_set:Npn \__keys_tmp:w #1
    {
      \cs_new:Npn \__keys_trim_spaces:n ##1
        {
          \exp_after:wN \__keys_trim_spaces_auxi:w \tl_to_str:n { / ##1 } /
            \s__keys_nil  \__keys_trim_spaces_auxi:w
            \s__keys_mark \__keys_trim_spaces_auxii:w
            #1 / #1
            \s__keys_nil  \__keys_trim_spaces_auxii:w
            \s__keys_mark \__keys_trim_spaces_auxiii:w
        }
    }
  \__keys_tmp:w { ~ }
\group_end:
\cs_new:Npn \__keys_trim_spaces_auxi:w #1 ~ / #2 \s__keys_nil #3
  {
```

```
    #3 #1 / #2 \s__keys_nil #3
  }
\cs_new:Npn \__keys_trim_spaces_auxii:w #1 / ~ #2 \s__keys_mark #3
  {
    #3 #1 / #2 \s__keys_mark #3
  }
\cs_new:Npn \__keys_trim_spaces_auxiii:w
    / #1 /
    \s__keys_nil  \__keys_trim_spaces_auxi:w
    \s__keys_mark \__keys_trim_spaces_auxii:w
    /
    \s__keys_nil  \__keys_trim_spaces_auxii:w
    \s__keys_mark \__keys_trim_spaces_auxiii:w
  {
    #1
  }
```

(*End of definition for* `\__keys_trim_spaces:n` *and others.*)

`\keys_if_exist_p:nn`
`\keys_if_exist:nnTF`

A utility for others to see if a key exists.

```
\prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
  {
    \cs_if_exist:cTF
      { \c__keys_code_root_str \__keys_trim_spaces:n { #1 / #2 } }
      { \prg_return_true: }
      { \prg_return_false: }
  }
\prg_generate_conditional_variant:Nnn \keys_if_exist:nn { ne } { p , T , F , TF }
```

(*End of definition for* `\keys_if_exist:nnTF`*. This function is documented on page 251.*)

`\keys_if_choice_exist_p:nnn`
`\keys_if_choice_exist:nnnTF`

Just an alternative view on `\keys_if_exist:nnTF`.

```
\prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3
  { p , T , F , TF }
  {
    \cs_if_exist:cTF
      { \c__keys_code_root_str \__keys_trim_spaces:n { #1 / #2 / #3 } }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End of definition for* `\keys_if_choice_exist:nnnTF`*. This function is documented on page 251.*)

`\keys_show:nn`
`\keys_log:nn`
`\__keys_show:Nnn`
`\__keys_show:n`
`\__keys_show:w`
`\__keys_show:Nw`

To show a key, show its code using a message.

```
\cs_new_protected:Npn \keys_show:nn
  { \__keys_show:Nnn \msg_show:nneeee }
\cs_new_protected:Npn \keys_log:nn
  { \__keys_show:Nnn \msg_log:nneeee }
\cs_new_protected:Npn \__keys_show:Nnn #1#2#3
  {
    #1 { keys } { show-key }
      { \__keys_trim_spaces:n { #2 / #3 } }
      {
        \keys_if_exist:nnT {#2} {#3}
          {
```

```
            \exp_args:Nnf \msg_show_item_unbraced:nn { code }
              {
                \exp_args:Ne \__keys_show:n
                  {
                    \exp_args:Nc \cs_replacement_spec:N
                    {
                      \c__keys_code_root_str
                      \__keys_trim_spaces:n { #2 / #3 }
                    }
                  }
              }
          }
      }
      { } { }
  }
\cs_new:Npe \__keys_show:n #1
  {
    \exp_not:N \__keys_show:w
      #1
      \tl_to_str:n { \__keys_precompile:n }
      #1
      \tl_to_str:n { \__keys_precompile:n }
      \exp_not:N \s__keys_stop
  }
\use:e
  {
    \cs_new:Npn \exp_not:N \__keys_show:w
      #1 \tl_to_str:n { \__keys_precompile:n }
      #2 \tl_to_str:n { \__keys_precompile:n }
      #3 \exp_not:N \s__keys_stop
  }
  {
    \tl_if_blank:nTF {#2}
      {#1}
      { \__keys_show:Nw #2 \s__keys_stop }
  }
\use:e
  {
    \cs_new:Npn \exp_not:N \__keys_show:Nw #1#2
      \c_right_brace_str \exp_not:N \s__keys_stop
  }
  {#2}
```

(*End of definition for* `\keys_show:nn` *and others. These functions are documented on page* *251*.)

65.8 Messages

For when there is a need to complain.

```
\msg_new:nnnn { keys } { bad-relative-key-path }
  { The~key~'#1'~is~not~inside~the~'#2'~path. }
  { The~key~'#1'~cannot~be~expressed~relative~to~path~'#2'. }
\msg_new:nnnn { keys } { boolean-values-only }
  { Key~'#1'~accepts~boolean~values~only. }
```

```
  { The~key~'#1'~only~accepts~the~values~'true'~and~'false'. }
\msg_new:nnnn { keys } { choice-unknown }
  { Key~'#1'~accepts~only~a~fixed~set~of~choices. }
  {
    The~key~'#1'~only~accepts~predefined~values,~
    and~'#2'~is~not~one~of~these.
  }
\msg_new:nnnn { keys } { unknown }
  { The~key~'#1'~is~unknown~and~is~being~ignored. }
  {
    The~module~'#2'~does~not~have~a~key~called~'#1'.\\
    Check~that~you~have~spelled~the~key~name~correctly.
  }
\msg_new:nnnn { keys } { nested-choice-key }
  { Attempt~to~define~'#1'~as~a~nested~choice~key. }
  {
    The~key~'#1'~cannot~be~defined~as~a~choice~as~the~parent~key~'#2'~is~
    itself~a~choice.
  }
\msg_new:nnnn { keys } { value-forbidden }
  { The~key~'#1'~does~not~take~a~value. }
  {
    The~key~'#1'~should~be~given~without~a~value.\\
    The~value~'#2'~was~present:~the~key~will~be~ignored.
  }
\msg_new:nnnn { keys } { value-required }
  { The~key~'#1'~requires~a~value. }
  {
    The~key~'#1'~must~have~a~value.\\
    No~value~was~present:~the~key~will~be~ignored.
  }
\msg_new:nnn { keys } { show-key }
  {
    The~key~#1~
    \tl_if_empty:nTF {#2}
      { is~undefined. }
      { has~the~properties: #2 . }
  }
\prop_gput:Nnn \g_msg_module_name_prop { keys } { LaTeX }
\prop_gput:Nnn \g_msg_module_type_prop { keys } { }
⟨/package⟩
```

Chapter 66

l3intarray implementation

```
⟨*package⟩
```

```
⟨@@=intarray⟩
```

There are two implementations for this module: One `\fontdimen` based one for more traditional TeX engines and a Lua based one for engines with Lua support.

Both versions do not allow negative array sizes.

```
⟨*tex⟩
\msg_new:nnn { kernel } { negative-array-size }
  { Size~of~array~may~not~be~negative:~#1 }
```

\l__intarray_loop_int

A loop index.

```
\int_new:N \l__intarray_loop_int
```

(*End of definition for* `\l__intarray_loop_int`.)

66.1 Lua implementation

First, let's look at the Lua variant:

We select the Lua version if the Lua helpers were defined. This can be detected by the presence of `\__intarray_gset_count:Nw`.

```
\cs_if_exist:NTF \__intarray_gset_count:Nw
  {
```

66.1.1 Allocating arrays

\g__intarray_table_int
\l__intarray_bad_index_int

Used to differentiate intarrays in Lua and to record an invalid index.

```
    \int_new:N \g__intarray_table_int
    \int_new:N \l__intarray_bad_index_int
⟨/tex⟩
```

(*End of definition for* `\g__intarray_table_int` *and* `\l__intarray_bad_index_int`.)

__intarray:w

Used as marker for intarrays in Lua. Followed by an unbraced number identifying the array and a single space. This format is used to make it easy to scan from Lua.

```
⟨*lua⟩
luacmd('__intarray:w', function()
  scan_int()
```

```
  tex.error'LaTeX Error: Isolated intarray ignored'
end, 'protected', 'global')
⟨/lua⟩
```

(*End of definition for* `\__intarray:w`.)

`\intarray_new:Nn` `\intarray_new:cn` `\__intarray_new:N`

Declare `#1` as a tokenlist with the scanmark and a unique number. Pass the array's size to the Lua helper. Every `intarray` must be global; it's enough to run this check in `\intarray_new:Nn`.

```
⟨*tex⟩
    \cs_new_protected:Npn \__intarray_new:N #1
      {
        \__kernel_chk_if_free_cs:N #1
        \int_gincr:N \g__intarray_table_int
        \cs_gset_nopar:Npe #1 { \__intarray:w \int_use:N \g__intarray_table_int \c_space_tl
      }
    \cs_new_protected:Npn \intarray_new:Nn #1#2
      {
        \__intarray_new:N #1
        \__intarray_gset_count:Nw #1 \int_eval:n {#2} \scan_stop:
        \int_compare:nNnT { \intarray_count:N #1 } < 0
          {
            \msg_error:nne { kernel } { negative-array-size }
              { \intarray_count:N #1 }
          }
      }
    \cs_generate_variant:Nn \intarray_new:Nn { c }
⟨/tex⟩
```

(*End of definition for* `\intarray_new:Nn` *and* `\__intarray_new:N`*. This function is documented on page 254.*)

Before we get to the first command implemented in Lua, we first need some definitions. Since `token.create` only works correctly if TeX has seen the tokens before, we first run a short TeX sequence to ensure that all relevant control sequences are known.

```
⟨*lua⟩

local scan_token = token.scan_token
local put_next = token.put_next
local intarray_marker = token_create_safe'__intarray:w'
local use_none = token_create_safe'use_none:n'
local use_i = token_create_safe'use:n'
local expand_after_scan_stop = {token_create_safe'exp_after:wN',
                                token_create_safe'scan_stop:'}
local comma = token_create(string.byte',')
```

`__intarray_table`

Internal helper to scan an intarray token, extract the associated Lua table and return an error if the input is invalid.

```
local __intarray_table do
  local tables = get_luadata and get_luadata'__intarray' or {[0] = {}}
  function __intarray_table()
    local t = scan_token()
    if t ~= intarray_marker then
      put_next(t)
      tex.error'LaTeX Error: intarray expected'
```

```
      return tables[0]
    end
    local i = scan_int()
    local current_table = tables[i]
    if current_table then return current_table end
    current_table = {}
    tables[i] = current_table
    return current_table
  end
```

Since in LaTeX this is loaded in the format, we want to preserve any intarrays which are created while format building for the actual run.

To do this, we use the `register_luadata` mechanism from l3luatex: Directly before the format get dumped, the following function gets invoked and serializes all existing tables into a string. This string gets compiled and dumped into the format and is made available at the beginning of regular runs as `get_luadata'@@'`.

```
  if register_luadata then
    register_luadata('__intarray', function()
      local t = "{[0]={},"
      for i=1, #tables do
        t = string.format("%s{%s},", t, table.concat(tables[i], ','))
      end
      return t .. "}"
    end)
  end
end
```

(*End of definition for* `__intarray_table`.)

`\intarray_count:N`
`\intarray_count:c`
`\__intarray_gset_count:Nw`

Set and get the size of an array. "Setting the size" means in this context that we add zeros until we reach the desired size.

```

local sprint = tex.sprint

luacmd('__intarray_gset_count:Nw', function()
  local t = __intarray_table()
  local n = scan_int()
  for i=#t+1, n do t[i] = 0 end
end, 'protected', 'global')

luacmd('intarray_count:N', function()
  sprint(-2, #__intarray_table())
end, 'global')
⟨/lua⟩
⟨*tex⟩
    \cs_generate_variant:Nn \intarray_count:N { c }
⟨/tex⟩
```

(*End of definition for* `\intarray_count:N` *and* `\__intarray_gset_count:Nw`. *This function is documented on page* 255.)

66.1.2 Array items

__intarray_gset:wF
__intarray_gset:w

The setter provided by Lua. The argument order somewhat emulates the `\fontdimen`: First the array index, followed by the intarray and then the new value. This has been chosen over a more conventional order to provide a delimiter for the numbers.

```
⟨*lua⟩
luacmd('__intarray_gset:wF', function()
  local i = scan_int()
  local t = __intarray_table()
  if t[i] then
    t[i] = scan_int()
    put_next(use_none)
  else
    tex.count.l__intarray_bad_index_int = i
    scan_int()
    put_next(use_i)
  end
end, 'protected', 'global')

luacmd('__intarray_gset:w', function()
  local i = scan_int()
  local t = __intarray_table()
  t[i] = scan_int()
end, 'protected', 'global')
⟨/lua⟩
```

(*End of definition for* `\__intarray_gset:wF` *and* `\__intarray_gset:w`.)

\intarray_gset:Nnn
\intarray_gset:cnn
__kernel_intarray_gset:Nnn

The `\__kernel_intarray_gset:Nnn` function does not use `\int_eval:n`, namely its arguments must be suitable for `\int_value:w`. The user version checks the position and value are within bounds.

```
⟨*tex⟩
  \cs_new_protected:Npn \__kernel_intarray_gset:Nnn #1#2#3
    { \__intarray_gset:w #2 #1 #3 \scan_stop: }
  \cs_new_protected:Npn \intarray_gset:Nnn #1#2#3
    {
      \__intarray_gset:wF \int_eval:n {#2} #1 \int_eval:n{#3}
        {
          \msg_error:nneee { kernel } { out-of-bounds }
            { \token_to_str:N #1 } { \int_use:N \l__intarray_bad_index_int } { \intarray_
        }
    }
  \cs_generate_variant:Nn \intarray_gset:Nnn { c }
⟨/tex⟩
```

(*End of definition for* `\intarray_gset:Nnn` *and* `\__kernel_intarray_gset:Nnn`. *This function is documented on page* 255.)

\intarray_gzero:N
\intarray_gzero:c

Set the appropriate array entry to zero. No bound checking needed.

```
⟨*lua⟩
luacmd('intarray_gzero:N', function()
  local t = __intarray_table()
  for i=1, #t do
    t[i] = 0
```

```
  end
end, 'global', 'protected')
⟨/lua⟩
⟨*tex⟩
    \cs_generate_variant:Nn \intarray_gzero:N { c }
⟨/tex⟩
```

(*End of definition for* `\intarray_gzero:N`. *This function is documented on page 254.*)

`\intarray_item:Nn`
`\intarray_item:cn`
`\__kernel_intarray_item:Nn`
`\__intarray_item:wF`
`\__intarray_item:w`

Get the appropriate entry and perform bound checks. The `\__kernel_intarray_item:Nn` function omits bound checks and omits `\int_eval:n`, namely its argument must be a TeX integer suitable for `\int_value:w`.

```
⟨*lua⟩
luacmd('__intarray_item:wF', function()
  local i = scan_int()
  local t = __intarray_table()
  local item = t[i]
  if item then
    put_next(use_none)
  else
    tex.l__intarray_bad_index_int = i
    put_next(use_i)
  end
  put_next(expand_after_scan_stop)
  scan_token()
  if item then
    sprint(-2, item)
  end
end, 'global')

luacmd('__intarray_item:w', function()
  local i = scan_int()
  local t = __intarray_table()
  sprint(-2, t[i])
end, 'global')
⟨/lua⟩
⟨*tex⟩
    \cs_new:Npn \__kernel_intarray_item:Nn #1#2
      { \__intarray_item:w #2 #1 }
    \cs_new:Npn \intarray_item:Nn #1#2
      {
        \__intarray_item:wF \int_eval:n {#2} #1
          {
            \msg_expandable_error:nnfff { kernel } { out-of-bounds }
              { \token_to_str:N #1 } { \int_use:N \l__intarray_bad_index_int } { \intarray_
            0
          }
      }
    \cs_generate_variant:Nn \intarray_item:Nn { c }
```

(*End of definition for* `\intarray_item:Nn` *and others. This function is documented on page 255.*)

`\intarray_rand_item:N`
`\intarray_rand_item:c`

Importantly, `\intarray_item:Nn` only evaluates its argument once.

```
    \cs_new:Npn \intarray_rand_item:N #1
```

```
      { \intarray_item:Nn #1 { \int_rand:n { \intarray_count:N #1 } } }
    \cs_generate_variant:Nn \intarray_rand_item:N { c }
```

(*End of definition for* `\intarray_rand_item:N`. *This function is documented on page* *255*.)

66.1.3 Working with contents of integer arrays

`\intarray_const_from_clist:Nn`
`\intarray_const_from_clist:cn`

We use the `\__kernel_intarray_gset:Nnn` which does not do bounds checking and instead automatically resizes the array. This is not implemented in Lua to ensure that the clist parsing is consistent with the clist module.

```
    \cs_new_protected:Npn \intarray_const_from_clist:Nn #1#2
      {
        \__intarray_new:N #1
        \int_zero:N \l__intarray_loop_int
        \clist_map_inline:nn {#2}
          {
            \int_incr:N \l__intarray_loop_int
            \__kernel_intarray_gset:Nnn #1 \l__intarray_loop_int { \int_eval:n {##1} } }
      }
    \cs_generate_variant:Nn \intarray_const_from_clist:Nn { c }
```

(*End of definition for* `\intarray_const_from_clist:Nn`. *This function is documented on page* *254*.)

`\__intarray_to_clist:Nn`
`\__intarray_to_clist:w`

The `\__intarray_to_clist:Nn` auxiliary allows to choose the delimiter and is also used in `\intarray_show:N`. Here we just pass the information to Lua and let `table.concat` do the actual work. We discard the category codes of the passed delimiter but this is not an issue since the delimiter is always just a comma or a comma and a space. In both cases `sprint(2, ...)` provides the right catcodes.

```
⟨/tex⟩
⟨*lua⟩
local concat = table.concat
luacmd('__intarray_to_clist:Nn', function()
  local t = __intarray_table()
  local sep = token.scan_string()
  sprint(-2, concat(t, sep))
end, 'global')
⟨/lua⟩
```

(*End of definition for* `\__intarray_to_clist:Nn` *and* `\__intarray_to_clist:w`.)

`\__kernel_intarray_range_to_clist:Nnn`
`\__intarray_range_to_clist:w`

Loop through part of the array.

```
⟨*tex⟩
    \cs_new:Npn \__kernel_intarray_range_to_clist:Nnn #1#2#3
      {
        \__intarray_range_to_clist:w #1
        \int_eval:n {#2} ~ \int_eval:n {#3} ~
      }
⟨/tex⟩
⟨*lua⟩
luacmd('__intarray_range_to_clist:w', function()
  local t = __intarray_table()
  local from = scan_int()
  local to = scan_int()
  sprint(-2, concat(t, ',', from, to))
```

```
end, 'global')
```
⟨/lua⟩

(*End of definition for* `\__kernel_intarray_range_to_clist:Nnn` *and* `\__intarray_range_to_clist:w`.)

`\__kernel_intarray_gset_range_from_clist:Nnn`
`\__intarray_gset_range:nNw`

Loop through part of the array. We allow additional commas at the end.

⟨*tex⟩
```
    \cs_new_protected:Npn \__kernel_intarray_gset_range_from_clist:Nnn #1#2#3
      {
        \__intarray_gset_range:w \int_eval:w #2 #1 #3 , , \scan_stop:
      }
```
⟨/tex⟩
⟨*lua⟩
```
luacmd('__intarray_gset_range:w', function()
  local from = scan_int()
  local t = __intarray_table()
  while true do
    local tok = scan_token()
    if tok == comma then
      repeat
        tok = scan_token()
      until tok ~= comma
      break
    else
      put_next(tok)
    end
    t[from] = scan_int()
    scan_token()
    from = from + 1
  end
  end, 'global', 'protected')
```
⟨/lua⟩

(*End of definition for* `\__kernel_intarray_gset_range_from_clist:Nnn` *and* `\__intarray_gset_range:nNw`.)

`\__intarray_gset_overflow_test:nw`

In order to allow some code sharing later we provide the `\__intarray_gset_overflow_test:nw` name here. It doesn't actually test anything since the Lua implementation accepts all integers which could be tested with `\tex_ifabsnum:D`.

⟨*tex⟩
```
    \cs_new_protected:Npn \__intarray_gset_overflow_test:nw #1
    {
    }
```

(*End of definition for* `\__intarray_gset_overflow_test:nw`.)

66.2 Font dimension based implementation

Go to the false branch of the conditional above.

```
  }
  {
```

66.2.1 Allocating arrays

__intarray_entry:w
__intarray_count:w

We use these primitives quite a lot in this module.

```
\cs_new_eq:NN \__intarray_entry:w \tex_fontdimen:D
\cs_new_eq:NN \__intarray_count:w \tex_hyphenchar:D
```

(*End of definition for* `\__intarray_entry:w` *and* `\__intarray_count:w`*.*)

\c__intarray_sp_dim

Used to convert integers to dimensions fast.

```
\dim_const:Nn \c__intarray_sp_dim { 1 sp }
```

(*End of definition for* `\c__intarray_sp_dim`*.*)

\g__intarray_font_int

Used to assign one font per array.

```
\int_new:N \g__intarray_font_int
```

(*End of definition for* `\g__intarray_font_int`*.*)

\intarray_new:Nn
\intarray_new:cn
__intarray_new:N

Declare `#1` to be a font (arbitrarily `cmr10` at a never-used size). Store the array's size as the `\hyphenchar` of that font and make sure enough `\fontdimen` are allocated, by setting the last one. Then clear any `\fontdimen` that `cmr10` starts with. It seems LuaTeX's `cmr10` has an extra `\fontdimen` parameter number 8 compared to other engines (for a math font we would replace 8 by 22 or some such). Every `intarray` must be global; it's enough to run this check in `\intarray_new:Nn`.

```
\cs_new_protected:Npn \__intarray_new:N #1
  {
    \__kernel_chk_if_free_cs:N #1
    \int_gincr:N \g__intarray_font_int
    \tex_global:D \tex_font:D #1
      = cmr10~at~ \g__intarray_font_int \c__intarray_sp_dim \scan_stop:
    \int_step_inline:nn { 8 }
      { \__kernel_intarray_gset:Nnn #1 {##1} \c_zero_int }
  }
\cs_new_protected:Npn \intarray_new:Nn #1#2
  {
    \__intarray_new:N #1
    \__intarray_count:w #1 = \int_eval:n {#2} \scan_stop:
    \int_compare:nNnT { \intarray_count:N #1 } < 0
      {
        \msg_error:nne { kernel } { negative-array-size }
          { \intarray_count:N #1 }
      }
    \int_compare:nNnT { \intarray_count:N #1 } > 0
      { \__kernel_intarray_gset:Nnn #1 { \intarray_count:N #1 } { 0 } }
  }
\cs_generate_variant:Nn \intarray_new:Nn { c }
```

(*End of definition for* `\intarray_new:Nn` *and* `\__intarray_new:N`*. This function is documented on page* *254*.)

\intarray_count:N
\intarray_count:c

Size of an array.

```
\cs_new:Npn \intarray_count:N #1 { \int_value:w \__intarray_count:w #1 }
\cs_generate_variant:Nn \intarray_count:N { c }
```

(*End of definition for* `\intarray_count:N`*. This function is documented on page* *255*.)

66.2.2 Array items

__intarray_signed_max_dim:n

Used when an item to be stored is larger than \c_max_dim in absolute value; it is replaced by ±\c_max_dim.

```
\cs_new:Npn \__intarray_signed_max_dim:n #1
  { \int_value:w \int_compare:nNnT {#1} < 0 { - } \c_max_dim }
```

(*End of definition for* __intarray_signed_max_dim:n.)

__intarray_bounds:NNnTF
__intarray_bounds_error:NNnw

The functions \intarray_gset:Nnn and \intarray_item:Nn share bounds checking. The T branch is used if #3 is within bounds of the array #2.

```
\cs_new:Npn \__intarray_bounds:NNnTF #1#2#3
  {
    \if_int_compare:w 1 > #3 \exp_stop_f:
      \__intarray_bounds_error:NNnw #1 #2 {#3}
    \else:
      \if_int_compare:w #3 > \intarray_count:N #2 \exp_stop_f:
        \__intarray_bounds_error:NNnw #1 #2 {#3}
      \fi:
    \fi:
    \use_i:nn
  }
\cs_new:Npn \__intarray_bounds_error:NNnw #1#2#3#4 \use_i:nn #5#6
  {
    #4
    #1 { kernel } { out-of-bounds }
      { \token_to_str:N #2 } {#3} { \intarray_count:N #2 }
    #6
  }
```

(*End of definition for* __intarray_bounds:NNnTF *and* __intarray_bounds_error:NNnw.)

\intarray_gset:Nnn
\intarray_gset:cnn
__kernel_intarray_gset:Nnn
__intarray_gset:Nnn
__intarray_gset_overflow:Nnn

Set the appropriate \fontdimen. The __kernel_intarray_gset:Nnn function does not use \int_eval:n, namely its arguments must be suitable for \int_value:w. The user version checks the position and value are within bounds.

```
\cs_new_protected:Npn \__kernel_intarray_gset:Nnn #1#2#3
  { \__intarray_entry:w #2 #1 #3 \c__intarray_sp_dim }
\cs_new_protected:Npn \intarray_gset:Nnn #1#2#3
  {
    \exp_after:wN \__intarray_gset:Nww
    \exp_after:wN #1
    \int_value:w \int_eval:n {#2} \exp_after:wN ;
    \int_value:w \int_eval:n {#3} ;
  }
\cs_generate_variant:Nn \intarray_gset:Nnn { c }
\cs_new_protected:Npn \__intarray_gset:Nww #1#2 ; #3 ;
  {
    \__intarray_bounds:NNnTF \msg_error:nneee #1 {#2}
      {
        \__intarray_gset_overflow_test:nw {#3}
        \__kernel_intarray_gset:Nnn #1 {#2} {#3}
      }
      { }
  }
```

```
    \cs_if_exist:NTF \tex_ifabsnum:D
      {
        \cs_new_protected:Npn \__intarray_gset_overflow_test:nw #1
          {
            \tex_ifabsnum:D #1 > \c_max_dim
              \exp_after:wN \__intarray_gset_overflow:NNnn
            \fi:
          }
      }
      {
        \cs_new_protected:Npn \__intarray_gset_overflow_test:nw #1
          {
            \if_int_compare:w \int_abs:n {#1} > \c_max_dim
              \exp_after:wN \__intarray_gset_overflow:NNnn
            \fi:
          }
      }
    \cs_new_protected:Npn \__intarray_gset_overflow:NNnn #1#2#3#4
      {
        \msg_error:nneeee { kernel } { overflow }
          { \token_to_str:N #2 } {#3} {#4} { \__intarray_signed_max_dim:n {#4} }
        #1 #2 {#3} { \__intarray_signed_max_dim:n {#4} }
      }
```

(*End of definition for* `\intarray_gset:Nnn` *and others. This function is documented on page 255.*)

`\intarray_gzero:N`
`\intarray_gzero:c`

Set the appropriate `\fontdimen` to zero. No bound checking needed. The `\prg_replicate:nn` possibly uses quite a lot of memory, but this is somewhat comparable to the size of the array, and it is much faster than an `\int_step_inline:nn` loop.

```
    \cs_new_protected:Npn \intarray_gzero:N #1
      {
        \int_zero:N \l__intarray_loop_int
        \prg_replicate:nn { \intarray_count:N #1 }
          {
            \int_incr:N \l__intarray_loop_int
            \__intarray_entry:w \l__intarray_loop_int #1 \c_zero_dim
          }
      }
    \cs_generate_variant:Nn \intarray_gzero:N { c }
```

(*End of definition for* `\intarray_gzero:N`. *This function is documented on page 254.*)

`\intarray_item:Nn`
`\intarray_item:cn`
`\__kernel_intarray_item:Nn`
`\__intarray_item:Nw`

Get the appropriate `\fontdimen` and perform bound checks. The `\__kernel_intarray_item:Nn` function omits bound checks and omits `\int_eval:n`, namely its argument must be a TeX integer suitable for `\int_value:w`.

```
    \cs_new:Npn \__kernel_intarray_item:Nn #1#2
      { \int_value:w \__intarray_entry:w #2 #1 }
    \cs_new:Npn \intarray_item:Nn #1#2
      {
        \exp_after:wN \__intarray_item:Nw
        \exp_after:wN #1
        \int_value:w \int_eval:n {#2} ;
      }
    \cs_generate_variant:Nn \intarray_item:Nn { c }
```

```
\cs_new:Npn \__intarray_item:Nw #1#2 ;
  {
    \__intarray_bounds:NNnTF \msg_expandable_error:nnfff #1 {#2}
      { \__kernel_intarray_item:Nn #1 {#2} }
      { 0 }
  }
```

(*End of definition for* `\intarray_item:Nn`*,* `\__kernel_intarray_item:Nn`*, and* `\__intarray_item:Nw`*. This function is documented on page 255.*)

`\intarray_rand_item:N`
`\intarray_rand_item:c`

Importantly, `\intarray_item:Nn` only evaluates its argument once.

```
\cs_new:Npn \intarray_rand_item:N #1
  { \intarray_item:Nn #1 { \int_rand:n { \intarray_count:N #1 } } }
\cs_generate_variant:Nn \intarray_rand_item:N { c }
```

(*End of definition for* `\intarray_rand_item:N`*. This function is documented on page 255.*)

66.2.3 Working with contents of integer arrays

`\intarray_const_from_clist:Nn`
`\intarray_const_from_clist:cn`
`\__intarray_const_from_clist:nN`

Similar to `\intarray_new:Nn` (which we don't use because when debugging is enabled that function checks the variable name starts with `g_`). We make use of the fact that TeX allows allocation of successive `\fontdimen` as long as no other font has been declared: no need to count the comma list items first. We need the code in `\intarray_gset:Nnn` that checks the item value is not too big, namely `\__intarray_gset_overflow_test:nw`, but not the code that checks bounds. At the end, set the size of the intarray.

```
\cs_new_protected:Npn \intarray_const_from_clist:Nn #1#2
  {
    \__intarray_new:N #1
    \int_zero:N \l__intarray_loop_int
    \clist_map_inline:nn {#2}
      { \exp_args:Nf \__intarray_const_from_clist:nN { \int_eval:n {##1} } #1 }
    \__intarray_count:w #1 \l__intarray_loop_int
  }
\cs_generate_variant:Nn \intarray_const_from_clist:Nn { c }
\cs_new_protected:Npn \__intarray_const_from_clist:nN #1#2
  {
    \int_incr:N \l__intarray_loop_int
    \__intarray_gset_overflow_test:nw {#1}
    \__kernel_intarray_gset:Nnn #2 \l__intarray_loop_int {#1}
  }
```

(*End of definition for* `\intarray_const_from_clist:Nn` *and* `\__intarray_const_from_clist:nN`*. This function is documented on page 254.*)

`\__intarray_to_clist:Nn`
`\__intarray_to_clist:w`

Loop through the array, putting a comma before each item. Remove the leading comma with `f`-expansion. We also use the auxiliary in `\intarray_show:N` with argument comma, space.

```
\cs_new:Npn \__intarray_to_clist:Nn #1#2
  {
    \int_compare:nNnF { \intarray_count:N #1 } = \c_zero_int
      {
        \exp_last_unbraced:Nf \use_none:n
          { \__intarray_to_clist:w 1 ; #1 {#2} \prg_break_point: }
      }
```

```
    }
  \cs_new:Npn \__intarray_to_clist:w #1 ; #2#3
    {
      \if_int_compare:w #1 > \__intarray_count:w #2
        \prg_break:n
      \fi:
      #3 \__kernel_intarray_item:Nn #2 {#1}
      \exp_after:wN \__intarray_to_clist:w
      \int_value:w \int_eval:w #1 + \c_one_int ; #2 {#3}
    }
```

(*End of definition for* `\__intarray_to_clist:Nn` *and* `\__intarray_to_clist:w`.)

`\__kernel_intarray_range_to_clist:Nnn`
`\__intarray_range_to_clist:ww`

Loop through part of the array.

```
  \cs_new:Npn \__kernel_intarray_range_to_clist:Nnn #1#2#3
    {
      \exp_last_unbraced:Nf \use_none:n
        {
          \exp_after:wN \__intarray_range_to_clist:ww
          \int_value:w \int_eval:w #2 \exp_after:wN ;
          \int_value:w \int_eval:w #3 ;
          #1 \prg_break_point:
        }
    }
  \cs_new:Npn \__intarray_range_to_clist:ww #1 ; #2 ; #3
    {
      \if_int_compare:w #1 > #2 \exp_stop_f:
        \prg_break:n
      \fi:
      , \__kernel_intarray_item:Nn #3 {#1}
      \exp_after:wN \__intarray_range_to_clist:ww
      \int_value:w \int_eval:w #1 + \c_one_int ; #2 ; #3
    }
```

(*End of definition for* `\__kernel_intarray_range_to_clist:Nnn` *and* `\__intarray_range_to_clist:ww`.)

`\__kernel_intarray_gset_range_from_clist:Nnn`
`\__intarray_gset_range:Nw`

Loop through part of the array.

```
  \cs_new_protected:Npn \__kernel_intarray_gset_range_from_clist:Nnn #1#2#3
    {
      \int_set:Nn \l__intarray_loop_int {#2}
      \__intarray_gset_range:Nw #1 #3 , , \prg_break_point:
    }
  \cs_new_protected:Npn \__intarray_gset_range:Nw #1 #2 ,
    {
      \if_catcode:w \scan_stop: \tl_to_str:n {#2} \scan_stop:
        \prg_break:n
      \fi:
      \__kernel_intarray_gset:Nnn #1 \l__intarray_loop_int {#2}
      \int_incr:N \l__intarray_loop_int
      \__intarray_gset_range:Nw #1
    }
```

(*End of definition for* `\__kernel_intarray_gset_range_from_clist:Nnn` *and* `\__intarray_gset_range:Nw`.)

```
}
```

66.3 Common parts

\intarray_if_exist_p:N
\intarray_if_exist_p:c
\intarray_if_exist:NTF
\intarray_if_exist:cTF

Copies of the cs functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \intarray_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \intarray_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End of definition for* \intarray_if_exist:NTF. *This function is documented on page 255.*)

\intarray_show:N
\intarray_show:c
\intarray_log:N
\intarray_log:c

Convert the list to a comma list (with spaces after each comma)

```
\cs_new_protected:Npn \intarray_show:N { \__intarray_show:NN \msg_show:nneeee }
\cs_generate_variant:Nn \intarray_show:N { c }
\cs_new_protected:Npn \intarray_log:N { \__intarray_show:NN \msg_log:nneeee }
\cs_generate_variant:Nn \intarray_log:N { c }
\cs_new_protected:Npn \__intarray_show:NN #1#2
  {
    \__kernel_chk_defined:NT #2
      {
        #1 { intarray } { show }
          { \token_to_str:N #2 }
          { \intarray_count:N #2 }
          { >~ \__intarray_to_clist:Nn #2 { , ~ } }
          { }
      }
  }
```

(*End of definition for* \intarray_show:N *and* \intarray_log:N. *These functions are documented on page 255.*)

```
</tex>
</package>
```

Chapter 67

l3fp implementation

Nothing to see here: everything is in the subfiles!

Chapter 68

l3fp-aux implementation

```
⟨*package⟩
⟨@@=fp⟩
```

68.1 Access to primitives

__fp_int_eval:w
__fp_int_eval_end:
__fp_int_to_roman:w

Largely for performance reasons, we need to directly access primitives rather than use `\int_eval:n`. This happens *a lot*, so we use private names. The same is true for `\romannumeral`, although it is used much less widely.

```
\cs_new_eq:NN \__fp_int_eval:w \tex_numexpr:D
\cs_new_eq:NN \__fp_int_eval_end: \scan_stop:
\cs_new_eq:NN \__fp_int_to_roman:w \tex_romannumeral:D
```

(*End of definition for* `\__fp_int_eval:w`, `\__fp_int_eval_end:`, *and* `\__fp_int_to_roman:w`.)

68.2 Internal representation

Internally, a floating point number ⟨*X*⟩ is a token list containing

`\s__fp \__fp_chk:w` ⟨*case*⟩ ⟨*sign*⟩ ⟨*body*⟩ `;`

Let us explain each piece separately.

Internal floating point numbers are used in expressions, and in this context are subject to `f`-expansion. They must leave a recognizable mark after `f`-expansion, to prevent the floating point number from being re-parsed. Thus, `\s__fp` is simply another name for `\relax`.

When used directly without an accessor function, floating points should produce an error: this is the role of `\__fp_chk:w`. We could make floating point variables be protected to prevent them from expanding under `e`/`x`-expansion, but it seems more convenient to treat them as a subcase of token list variables.

The (decimal part of the) IEEE-754-2008 standard requires the format to be able to represent special floating point numbers besides the usual positive and negative cases. We distinguish the various possibilities by their ⟨*case*⟩, which is a single digit:

0 zeros: `+0` and `-0`,

1 "normal" numbers (positive and negative),

Table 3: Internal representation of floating point numbers.

Representation	Meaning
0 0 \s__fp_... ;	Positive zero.
0 2 \s__fp_... ;	Negative zero.
1 0 {⟨*exponent*⟩} {⟨X_1⟩} {⟨X_2⟩} {⟨X_3⟩} {⟨X_4⟩} ;	Positive floating point.
1 2 {⟨*exponent*⟩} {⟨X_1⟩} {⟨X_2⟩} {⟨X_3⟩} {⟨X_4⟩} ;	Negative floating point.
2 0 \s__fp_... ;	Positive infinity.
2 2 \s__fp_... ;	Negative infinity.
3 1 \s__fp_... ;	Quiet nan.
3 1 \s__fp_... ;	Signalling nan.

2 infinities: `+inf` and `-inf`,

3 quiet and signalling `nan`.

The ⟨*sign*⟩ is `0` (positive) or `2` (negative), except in the case of `nan`, which have ⟨*sign*⟩ $=$ 1. This ensures that changing the ⟨*sign*⟩ digit to $2 -$ ⟨*sign*⟩ is exactly equivalent to changing the sign of the number.

Special floating point numbers have the form

`\s__fp \__fp_chk:w` ⟨*case*⟩ ⟨*sign*⟩ `\s__fp_... ;`

where `\s__fp_...` is a scan mark carrying information about how the number was formed (useful for debugging).

Normal floating point numbers (⟨*case*⟩ $= 1$) have the form

`\s__fp \__fp_chk:w` 1 ⟨*sign*⟩ {⟨*exponent*⟩} {⟨X_1⟩} {⟨X_2⟩} {⟨X_3⟩} {⟨X_4⟩} ;

Here, the ⟨*exponent*⟩ is an integer, between -10000 and 10000. The body consists in four blocks of exactly 4 digits, $0000 \leq \langle X_i \rangle \leq 9999$, and the floating point is

$$(-1)^{\langle sign \rangle/2} \langle X_1\rangle\langle X_2\rangle\langle X_3\rangle\langle X_4\rangle \cdot 10^{\langle exponent \rangle - 16}$$

where we have concatenated the 16 digits. Currently, floating point numbers are normalized such that the ⟨*exponent*⟩ is minimal, in other words, $1000 \leq \langle X_1 \rangle \leq 9999$.

Calculations are done in base 10000, *i.e.* one myriad.

68.3 Using arguments and semicolons

`\__fp_use_none_stop_f:n` This function removes an argument (typically a digit) and replaces it by `\exp_stop_f:`, a marker which stops `f`-type expansion.

```
\cs_new:Npn \__fp_use_none_stop_f:n #1 { \exp_stop_f: }
```

(*End of definition for* `\__fp_use_none_stop_f:n`.)

`\__fp_use_s:n`
`\__fp_use_s:nn` Those functions place a semicolon after one or two arguments (typically digits).

```
\cs_new:Npn \__fp_use_s:n #1 { #1; }
\cs_new:Npn \__fp_use_s:nn #1#2 { #1#2; }
```

(*End of definition for* `\__fp_use_s:n` *and* `\__fp_use_s:nn`.)

__fp_use_none_until_s:w
__fp_use_i_until_s:nw
__fp_use_ii_until_s:nnw

Those functions select specific arguments among a set of arguments delimited by a semicolon.

```
\cs_new:Npn \__fp_use_none_until_s:w #1; { }
\cs_new:Npn \__fp_use_i_until_s:nw #1#2; {#1}
\cs_new:Npn \__fp_use_ii_until_s:nnw #1#2#3; {#2}
```

(*End of definition for* __fp_use_none_until_s:w *,* __fp_use_i_until_s:nw *, and* __fp_use_ii_until_s:nnw*.*)

__fp_reverse_args:Nww

Many internal functions take arguments delimited by semicolons, and it is occasionally useful to swap two such arguments.

```
\cs_new:Npn \__fp_reverse_args:Nww #1 #2; #3; { #1 #3; #2; }
```

(*End of definition for* __fp_reverse_args:Nww*.*)

__fp_rrot:www

Rotate three arguments delimited by semicolons. This is the inverse (or the square) of the Forth primitive `ROT`, hence the name.

```
\cs_new:Npn \__fp_rrot:www #1; #2; #3; { #2; #3; #1; }
```

(*End of definition for* __fp_rrot:www*.*)

__fp_use_i:ww
__fp_use_i:www

Many internal functions take arguments delimited by semicolons, and it is occasionally useful to remove one or two such arguments.

```
\cs_new:Npn \__fp_use_i:ww #1; #2; { #1; }
\cs_new:Npn \__fp_use_i:www #1; #2; #3; { #1; }
```

(*End of definition for* __fp_use_i:ww *and* __fp_use_i:www*.*)

68.4 Constants, and structure of floating points

__fp_misused:n

This receives a floating point object (floating point number or tuple) and generates an error stating that it was misused. This is called when for instance an `fp` variable is left in the input stream and its contents reach TeX's stomach.

```
\cs_new_protected:Npn \__fp_misused:n #1
  { \msg_error:nne { fp } { misused } { \fp_to_tl:n {#1} } }
```

(*End of definition for* __fp_misused:n*.*)

\s__fp
__fp_chk:w

Floating points numbers all start with `\s__fp \__fp_chk:w`, where `\s__fp` is equal to the TeX primitive `\relax`, and `\__fp_chk:w` is protected. The rest of the floating point number is made of characters (or `\relax`). This ensures that nothing expands under `f`-expansion, nor under `e`/`x`-expansion. However, when typeset, `\s__fp` does nothing, and `\__fp_chk:w` is expanded. We define `\__fp_chk:w` to produce an error.

```
\scan_new:N \s__fp
\cs_new_protected:Npn \__fp_chk:w #1 ;
  { \__fp_misused:n { \s__fp \__fp_chk:w #1 ; } }
```

(*End of definition for* \s__fp *and* __fp_chk:w*.*)

\s__fp_expr_mark
\s__fp_expr_stop

Aliases of `\tex_relax:D`, used to terminate expressions.

```
\scan_new:N \s__fp_expr_mark
\scan_new:N \s__fp_expr_stop
```

(*End of definition for* \s__fp_expr_mark *and* \s__fp_expr_stop*.*)

\s__fp_mark
\s__fp_stop

Generic scan marks used throughout the module.

```
\scan_new:N \s__fp_mark
\scan_new:N \s__fp_stop
```

(*End of definition for* `\s__fp_mark` *and* `\s__fp_stop`*.*)

__fp_use_i_delimit_by_s_stop:nw

Functions to gobble up to a scan mark.

```
\cs_new:Npn \__fp_use_i_delimit_by_s_stop:nw #1 #2 \s__fp_stop {#1}
```

(*End of definition for* `\__fp_use_i_delimit_by_s_stop:nw`*.*)

\s__fp_invalid
\s__fp_underflow
\s__fp_overflow
\s__fp_division
\s__fp_exact

A couple of scan marks used to indicate where special floating point numbers come from.

```
\scan_new:N \s__fp_invalid
\scan_new:N \s__fp_underflow
\scan_new:N \s__fp_overflow
\scan_new:N \s__fp_division
\scan_new:N \s__fp_exact
```

(*End of definition for* `\s__fp_invalid` *and others.*)

\c_zero_fp
\c_minus_zero_fp
\c_inf_fp
\c_minus_inf_fp
\c_nan_fp

The special floating points. We define the floating points here as "exact".

```
\tl_const:Nn \c_zero_fp       { \s__fp \__fp_chk:w 0 0 \s__fp_exact ; }
\tl_const:Nn \c_minus_zero_fp { \s__fp \__fp_chk:w 0 2 \s__fp_exact ; }
\tl_const:Nn \c_inf_fp        { \s__fp \__fp_chk:w 2 0 \s__fp_exact ; }
\tl_const:Nn \c_minus_inf_fp  { \s__fp \__fp_chk:w 2 2 \s__fp_exact ; }
\tl_const:Nn \c_nan_fp        { \s__fp \__fp_chk:w 3 1 \s__fp_exact ; }
```

(*End of definition for* `\c_zero_fp` *and others. These variables are documented on page 268.*)

\c__fp_prec_int
\c__fp_half_prec_int
\c__fp_block_int

The number of digits of floating points.

```
\int_const:Nn \c__fp_prec_int { 16 }
\int_const:Nn \c__fp_half_prec_int { 8 }
\int_const:Nn \c__fp_block_int { 4 }
```

(*End of definition for* `\c__fp_prec_int`*,* `\c__fp_half_prec_int`*, and* `\c__fp_block_int`*.*)

\c__fp_myriad_int

Blocks have 4 digits so this integer is useful.

```
\int_const:Nn \c__fp_myriad_int { 10000 }
```

(*End of definition for* `\c__fp_myriad_int`*.*)

\c__fp_minus_min_exponent_int
\c__fp_max_exponent_int

Normal floating point numbers have an exponent between − `minus_min_exponent` and `max_exponent` inclusive. Larger numbers are rounded to $\pm\infty$. Smaller numbers are rounded to ± 0. It would be more natural to define a `min_exponent` with the opposite sign but that would waste one TeX count.

```
\int_const:Nn \c__fp_minus_min_exponent_int { 10000 }
\int_const:Nn \c__fp_max_exponent_int { 10000 }
```

(*End of definition for* `\c__fp_minus_min_exponent_int` *and* `\c__fp_max_exponent_int`*.*)

\c__fp_max_exp_exponent_int

If a number's exponent is larger than that, its exponential overflows/underflows.

```
\int_const:Nn \c__fp_max_exp_exponent_int { 5 }
```

(*End of definition for* `\c__fp_max_exp_exponent_int`*.*)

\c__fp_overflowing_fp A floating point number that is bigger than all normal floating point numbers. This replaces infinities when converting to formats that do not support infinities.

```
\tl_const:Ne \c__fp_overflowing_fp
  {
    \s__fp \__fp_chk:w 1 0
      { \int_eval:n { \c__fp_max_exponent_int + 1 } }
      {1000} {0000} {0000} {0000} ;
  }
```

(*End of definition for* `\c__fp_overflowing_fp`*.*)

__fp_zero_fp:N
__fp_inf_fp:N

In case of overflow or underflow, we have to output a zero or infinity with a given sign.

```
\cs_new:Npn \__fp_zero_fp:N #1
  { \s__fp \__fp_chk:w 0 #1 \s__fp_underflow ; }
\cs_new:Npn \__fp_inf_fp:N #1
  { \s__fp \__fp_chk:w 2 #1 \s__fp_overflow ; }
```

(*End of definition for* `\__fp_zero_fp:N` *and* `\__fp_inf_fp:N`*.*)

__fp_exponent:w For normal numbers, the function expands to the exponent, otherwise to 0. This is used in l3str-format.

```
\cs_new:Npn \__fp_exponent:w \s__fp \__fp_chk:w #1
  {
    \if_meaning:w 1 #1
      \exp_after:wN \__fp_use_ii_until_s:nnw
    \else:
      \exp_after:wN \__fp_use_i_until_s:nw
      \exp_after:wN 0
    \fi:
  }
```

(*End of definition for* `\__fp_exponent:w`*.*)

__fp_neg_sign:N When appearing in an integer expression or after `\int_value:w`, this expands to the sign opposite to `#1`, namely 0 (positive) is turned to 2 (negative), 1 (`nan`) to 1, and 2 to 0.

```
\cs_new:Npn \__fp_neg_sign:N #1
  { \__fp_int_eval:w 2 - #1 \__fp_int_eval_end: }
```

(*End of definition for* `\__fp_neg_sign:N`*.*)

__fp_kind:w Expands to 0 for zeros, 1 for normal floating point numbers, 2 for infinities, 3 for `nan`, 4 for tuples.

```
\cs_new:Npn \__fp_kind:w #1
  {
    \__fp_if_type_fp:NTwFw
      #1 \__fp_use_ii_until_s:nnw
      \s__fp { \__fp_use_i_until_s:nw 4 }
      \s__fp_stop
  }
```

(*End of definition for* `\__fp_kind:w`*.*)

68.5 Overflow, underflow, and exact zero

__fp_sanitize:Nw
__fp_sanitize:wN
__fp_sanitize_zero:w

Expects the sign and the exponent in some order, then the significand (which we don't touch). Outputs the corresponding floating point number, possibly underflowed to ± 0 or overflowed to $\pm\infty$. The functions `\__fp_underflow:w` and `\__fp_overflow:w` are defined in l3fp-traps.

```
\cs_new:Npn \__fp_sanitize:Nw #1 #2;
  {
    \if_case:w
        \if_int_compare:w #2 > \c__fp_max_exponent_int 1 ~ \else:
        \if_int_compare:w #2 < - \c__fp_minus_min_exponent_int 2 ~ \else:
        \if_meaning:w 1 #1 3 ~ \fi: \fi: \fi: 0 ~
    \or: \exp_after:wN \__fp_overflow:w
    \or: \exp_after:wN \__fp_underflow:w
    \or: \exp_after:wN \__fp_sanitize_zero:w
    \fi:
    \s__fp \__fp_chk:w 1 #1 {#2}
  }
\cs_new:Npn \__fp_sanitize:wN #1; #2 { \__fp_sanitize:Nw #2 #1; }
\cs_new:Npn \__fp_sanitize_zero:w \s__fp \__fp_chk:w #1 #2 #3;
  { \c_zero_fp }
```

(*End of definition for* `\__fp_sanitize:Nw`, `\__fp_sanitize:wN`, *and* `\__fp_sanitize_zero:w`.)

68.6 Expanding after a floating point number

__fp_exp_after_o:w
__fp_exp_after_f:nw

`\__fp_exp_after_o:w` ⟨*floating point*⟩
`\__fp_exp_after_f:nw` {⟨*tokens*⟩} ⟨*floating point*⟩

Places ⟨*tokens*⟩ (empty in the case of `\__fp_exp_after_o:w`) between the ⟨*floating point*⟩ and the following tokens, then hits those tokens with `o` or `f`-expansion, and leaves the floating point number unchanged.

We first distinguish normal floating points, which have a significand, from the much simpler special floating points.

```
\cs_new:Npn \__fp_exp_after_o:w \s__fp \__fp_chk:w #1
  {
    \if_meaning:w 1 #1
      \exp_after:wN \__fp_exp_after_normal:nNNw
    \else:
      \exp_after:wN \__fp_exp_after_special:nNNw
    \fi:
    { }
    #1
  }
\cs_new:Npn \__fp_exp_after_f:nw #1 \s__fp \__fp_chk:w #2
  {
    \if_meaning:w 1 #2
      \exp_after:wN \__fp_exp_after_normal:nNNw
    \else:
      \exp_after:wN \__fp_exp_after_special:nNNw
    \fi:
    { \exp:w \exp_end_continue_f:w #1 }
    #2
```

```
  }
```

(*End of definition for* `\__fp_exp_after_o:w` *and* `\__fp_exp_after_f:nw`.)

`\__fp_exp_after_special:nNNw`

`\__fp_exp_after_special:nNNw {⟨after⟩} ⟨case⟩ ⟨sign⟩ ⟨scan mark⟩ ;`

Special floating point numbers are easy to jump over since they contain few tokens.

```
\cs_new:Npn \__fp_exp_after_special:nNNw #1#2#3#4;
  {
    \exp_after:wN \s__fp
    \exp_after:wN \__fp_chk:w
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN #4
    \exp_after:wN ;
    #1
  }
```

(*End of definition for* `\__fp_exp_after_special:nNNw`.)

`\__fp_exp_after_normal:nNNw`

For normal floating point numbers, life is slightly harder, since we have many tokens to jump over. Here it would be slightly better if the digits were not braced but instead were delimited arguments (for instance delimited by `,`). That may be changed some day.

```
\cs_new:Npn \__fp_exp_after_normal:nNNw #1 1 #2 #3 #4#5#6#7;
  {
    \exp_after:wN \__fp_exp_after_normal:Nwwwww
    \exp_after:wN #2
    \int_value:w #3   \exp_after:wN ;
    \int_value:w 1 #4 \exp_after:wN ;
    \int_value:w 1 #5 \exp_after:wN ;
    \int_value:w 1 #6 \exp_after:wN ;
    \int_value:w 1 #7 \exp_after:wN ; #1
  }
\cs_new:Npn \__fp_exp_after_normal:Nwwwww
    #1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;
  { \s__fp \__fp_chk:w 1 #1 {#2} {#3} {#4} {#5} {#6} ; }
```

(*End of definition for* `\__fp_exp_after_normal:nNNw`.)

68.7 Other floating point types

`\s__fp_tuple`
`\__fp_tuple_chk:w`
`\c__fp_empty_tuple_fp`

Floating point tuples take the form `\s__fp_tuple \__fp_tuple_chk:w { ⟨fp 1⟩ ⟨fp 2⟩ ... } ;` where each ⟨*fp*⟩ is a floating point number or tuple, hence ends with `;` itself. When a tuple is typeset, `\__fp_tuple_chk:w` produces an error, just like usual floating point numbers. Tuples may have zero or one element.

```
\scan_new:N \s__fp_tuple
\cs_new_protected:Npn \__fp_tuple_chk:w #1 ;
  { \__fp_misused:n { \s__fp_tuple \__fp_tuple_chk:w #1 ; } }
\tl_const:Nn \c__fp_empty_tuple_fp
  { \s__fp_tuple \__fp_tuple_chk:w { } ; }
```

(*End of definition for* `\s__fp_tuple`, `\__fp_tuple_chk:w`, *and* `\c__fp_empty_tuple_fp`.)

__fp_tuple_count:w
__fp_array_count:n
__fp_tuple_count_loop:Nw

Count the number of items in a tuple of floating points by counting semicolons. The technique is very similar to `\tl_count:n`, but with the loop built-in. Checking for the end of the loop is done with the `\use_none:n #1` construction.

```
\cs_new:Npn \__fp_array_count:n #1
  { \__fp_tuple_count:w \s__fp_tuple \__fp_tuple_chk:w {#1} ; }
\cs_new:Npn \__fp_tuple_count:w \s__fp_tuple \__fp_tuple_chk:w #1 ;
  {
    \int_value:w \__fp_int_eval:w 0
      \__fp_tuple_count_loop:Nw #1 { ? \prg_break: } ;
      \prg_break_point:
    \__fp_int_eval_end:
  }
\cs_new:Npn \__fp_tuple_count_loop:Nw #1#2;
  { \use_none:n #1 + 1 \__fp_tuple_count_loop:Nw }
```

(*End of definition for* `\__fp_tuple_count:w`*,* `\__fp_array_count:n`*, and* `\__fp_tuple_count_loop:Nw`*.*)

__fp_if_type_fp:NTwFw

Used as `\__fp_if_type_fp:NTwFw` ⟨*marker*⟩ {⟨*true code*⟩} `\s__fp` {⟨*false code*⟩} `\s__fp_stop`, this test whether the ⟨*marker*⟩ is `\s__fp` or not and runs the appropriate ⟨*code*⟩. The very unusual syntax is for optimization purposes as that function is used for all floating point operations.

```
\cs_new:Npn \__fp_if_type_fp:NTwFw #1 \s__fp #2 #3 \s__fp_stop {#2}
```

(*End of definition for* `\__fp_if_type_fp:NTwFw`*.*)

__fp_array_if_all_fp:nTF
__fp_array_if_all_fp_loop:w

True if all items are floating point numbers. Used for `min`.

```
\cs_new:Npn \__fp_array_if_all_fp:nTF #1
  {
    \__fp_array_if_all_fp_loop:w #1 { \s__fp \prg_break: } ;
    \prg_break_point: \use_i:nn
  }
\cs_new:Npn \__fp_array_if_all_fp_loop:w #1#2 ;
  {
    \__fp_if_type_fp:NTwFw
      #1 \__fp_array_if_all_fp_loop:w
      \s__fp { \prg_break:n \use_iii:nnn }
      \s__fp_stop
  }
```

(*End of definition for* `\__fp_array_if_all_fp:nTF` *and* `\__fp_array_if_all_fp_loop:w`*.*)

__fp_type_from_scan:N
__fp_type_from_scan_other:N
__fp_type_from_scan:w

Used as `\__fp_type_from_scan:N` ⟨*token*⟩. Grabs the pieces of the stringified ⟨*token*⟩ which lies after the first `s__fp`. If the ⟨*token*⟩ does not contain that string, the result is `_?`.

```
\cs_new:Npn \__fp_type_from_scan:N #1
  {
    \__fp_if_type_fp:NTwFw
      #1 { }
      \s__fp { \__fp_type_from_scan_other:N #1 }
      \s__fp_stop
  }
\cs_new:Npe \__fp_type_from_scan_other:N #1
  {
    \exp_not:N \exp_after:wN \exp_not:N \__fp_type_from_scan:w
```

```
    \exp_not:N \token_to_str:N #1 \s__fp_mark
      \tl_to_str:n { s__fp _? } \s__fp_mark \s__fp_stop
  }
\exp_last_unbraced:NNNNo
  \cs_new:Npn \__fp_type_from_scan:w #1
    { \tl_to_str:n { s__fp } } #2 \s__fp_mark #3 \s__fp_stop {#2}
```

(*End of definition for* `\__fp_type_from_scan:N`*,* `\__fp_type_from_scan_other:N`*, and* `\__fp_type_from_scan:w`*.*)

`\__fp_change_func_type:NNN`
`\__fp_change_func_type_aux:w`
`\__fp_change_func_type_chk:NNN`

Arguments are ⟨*type marker*⟩ ⟨*function*⟩ ⟨*recovery*⟩. This gives the function obtained by placing the type after `@@`. If the function is not defined then ⟨*recovery*⟩ ⟨*function*⟩ is used instead; however that test is not run when the ⟨*type marker*⟩ is `\s__fp`.

```
\cs_new:Npn \__fp_change_func_type:NNN #1#2#3
  {
    \__fp_if_type_fp:NTwFw
      #1 #2
      \s__fp
        {
          \exp_after:wN \__fp_change_func_type_chk:NNN
          \cs:w
            __fp \__fp_type_from_scan_other:N #1
            \exp_after:wN \__fp_change_func_type_aux:w \token_to_str:N #2
          \cs_end:
          #2 #3
        }
      \s__fp_stop
  }
\exp_last_unbraced:NNNNo
  \cs_new:Npn \__fp_change_func_type_aux:w #1 { \tl_to_str:n { __fp } } { }
\cs_new:Npn \__fp_change_func_type_chk:NNN #1#2#3
  {
    \if_meaning:w \scan_stop: #1
      \exp_after:wN #3 \exp_after:wN #2
    \else:
      \exp_after:wN #1
    \fi:
  }
```

(*End of definition for* `\__fp_change_func_type:NNN`*,* `\__fp_change_func_type_aux:w`*, and* `\__fp_change_func_type_chk:NNN`*.*)

`\__fp_exp_after_any_f:Nnw`
`\__fp_exp_after_any_f:nw`
`\__fp_exp_after_expr_stop_f:nw`

The `Nnw` function simply dispatches to the appropriate `\__fp_exp_after..._f:nw` with "..." (either empty or `_`⟨*type*⟩) extracted from `#1`, which should start with `\s__fp`. If it doesn't start with `\s__fp` the function `\__fp_exp_after_?_f:nw` defined in l3fp-parse gives an error; another special ⟨*type*⟩ is `stop`, useful for loops, see below. The `nw` function has an important optimization for floating points numbers; it also fetches its type marker `#2` from the floating point.

```
\cs_new:Npn \__fp_exp_after_any_f:Nnw #1
  { \cs:w __fp_exp_after \__fp_type_from_scan_other:N #1 _f:nw \cs_end: }
\cs_new:Npn \__fp_exp_after_any_f:nw #1#2
  {
    \__fp_if_type_fp:NTwFw
      #2 \__fp_exp_after_f:nw
```

```
      \s__fp { \__fp_exp_after_any_f:Nnw #2 }
      \s__fp_stop
    {#1} #2
  }
\cs_new_eq:NN \__fp_exp_after_expr_stop_f:nw \use_none:nn
```

(*End of definition for* `\__fp_exp_after_any_f:Nnw`*,* `\__fp_exp_after_any_f:nw`*, and* `\__fp_exp_after_expr_stop_f:nw`*.*)

`\__fp_exp_after_tuple_o:w`
`\__fp_exp_after_tuple_f:nw`
`\__fp_exp_after_array_f:w`

The loop works by using the `n` argument of `\__fp_exp_after_any_f:nw` to place the loop macro after the next item in the tuple and expand it.

`\__fp_exp_after_array_f:w`
$\langle fp_1 \rangle$;
...
$\langle fp_n \rangle$;
`\s__fp_expr_stop`

```
\cs_new:Npn \__fp_exp_after_tuple_o:w
  { \__fp_exp_after_tuple_f:nw { \exp_after:wN \exp_stop_f: } }
\cs_new:Npn \__fp_exp_after_tuple_f:nw
  #1 \s__fp_tuple \__fp_tuple_chk:w #2 ;
  {
    \exp_after:wN \s__fp_tuple
    \exp_after:wN \__fp_tuple_chk:w
    \exp_after:wN {
      \exp:w \exp_end_continue_f:w
      \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
    \exp_after:wN }
    \exp_after:wN ;
    \exp:w \exp_end_continue_f:w #1
  }
\cs_new:Npn \__fp_exp_after_array_f:w
  { \__fp_exp_after_any_f:nw { \__fp_exp_after_array_f:w } }
```

(*End of definition for* `\__fp_exp_after_tuple_o:w`*,* `\__fp_exp_after_tuple_f:nw`*, and* `\__fp_exp_after_array_f:w`*.*)

68.8 Packing digits

When a positive integer `#1` is known to be less than 10^8, the following trick splits it into two blocks of 4 digits, padding with zeros on the left.

```
\cs_new:Npn \pack:NNNNNw #1 #2#3#4#5 #6; { {#2#3#4#5} {#6} }
\exp_after:wN \pack:NNNNNw
  \__fp_int_value:w \__fp_int_eval:w 1 0000 0000 + #1 ;
```

The idea is that adding 10^8 to the number ensures that it has exactly 9 digits, and can then easily find which digits correspond to what position in the number. Of course, this can be modified for any number of digits less or equal to 9 (we are limited by TeX's integers). This method is very heavily relied upon in `l3fp-basics`.

More specifically, the auxiliary inserts `+ #1#2#3#4#5 ; {#6}`, which allows us to compute several blocks of 4 digits in a nested manner, performing carries on the fly. Say we want to compute $1\,2345 \times 6677\,8899$. With simplified names, we would do

```
\exp_after:wN \post_processing:w
\__fp_int_value:w \__fp_int_eval:w - 5 0000
  \exp_after:wN \pack:NNNNNw
  \__fp_int_value:w \__fp_int_eval:w 4 9995 0000
    + 12345 * 6677
    \exp_after:wN \pack:NNNNNw
    \__fp_int_value:w \__fp_int_eval:w 5 0000 0000
      + 12345 * 8899 ;
```

The `\exp_after:wN` triggers `\int_value:w \__fp_int_eval:w`, which starts a first computation, whose initial value is $-5\,0000$ (the "leading shift"). In that computation appears an `\exp_after:wN`, which triggers the nested computation `\int_value:w \__fp_int_eval:w` with starting value $4\,9995\,0000$ (the "middle shift"). That, in turn, expands `\exp_after:wN` which triggers the third computation. The third computation's value is $5\,0000\,0000 + 12345 \times 8899$, which has 9 digits. Adding $5 \cdot 10^8$ to the product allowed us to know how many digits to expect as long as the numbers to multiply are not too big; it also works to some extent with negative results. The `pack` function puts the last 4 of those 9 digits into a brace group, moves the semi-colon delimiter, and inserts a `+`, which combines the carry with the previous computation. The shifts nicely combine into $5\,0000\,0000/10^4 + 4\,9995\,0000 = 5\,0000\,0000$. As long as the operands are in some range, the result of this second computation has 9 digits. The corresponding `pack` function, expanded after the result is computed, braces the last 4 digits, and leaves `+` ⟨*5 digits*⟩ for the initial computation. The "leading shift" cancels the combination of the other shifts, and the `\post_processing:w` takes care of packing the last few digits.

Admittedly, this is quite intricate. It is probably the key in making l3fp as fast as other pure TEX floating point units despite its increased precision. In fact, this is used so much that we provide different sets of packing functions and shifts, depending on ranges of input.

`\__fp_pack:NNNNNw`
`\c__fp_trailing_shift_int`
`\c__fp_middle_shift_int`
`\c__fp_leading_shift_int`

This set of shifts allows for computations involving results in the range $[-4\cdot 10^8, 5\cdot 10^8-1]$. Shifted values all have exactly 9 digits.

```
\int_const:Nn \c__fp_leading_shift_int  { - 5 0000 }
\int_const:Nn \c__fp_middle_shift_int   { 5 0000 *  9999 }
\int_const:Nn \c__fp_trailing_shift_int { 5 0000 * 10000 }
\cs_new:Npn \__fp_pack:NNNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }
```

(*End of definition for* `\__fp_pack:NNNNNw` *and others.*)

`\__fp_pack_big:NNNNNNw`
`\c__fp_big_trailing_shift_int`
`\c__fp_big_middle_shift_int`
`\c__fp_big_leading_shift_int`

This set of shifts allows for computations involving results in the range $[-5\cdot 10^8, 6\cdot 10^8-1]$ (actually a bit more). Shifted values all have exactly 10 digits. Note that the upper bound is due to TEX's limit of $2^{31}-1$ on integers. The shifts are chosen to be roughly the mid-point of 10^9 and 2^{31}, the two bounds on 10-digit integers in TEX.

```
\int_const:Nn \c__fp_big_leading_shift_int  { - 15 2374 }
\int_const:Nn \c__fp_big_middle_shift_int   { 15 2374 *  9999 }
\int_const:Nn \c__fp_big_trailing_shift_int { 15 2374 * 10000 }
\cs_new:Npn \__fp_pack_big:NNNNNNw #1#2 #3#4#5#6 #7;
  { + #1#2#3#4#5#6 ; {#7} }
```

(*End of definition for* `\__fp_pack_big:NNNNNNw` *and others.*)

__fp_pack_Bigg:NNNNNNw
\c__fp_Bigg_trailing_shift_int
\c__fp_Bigg_middle_shift_int
\c__fp_Bigg_leading_shift_int

This set of shifts allows for computations with results in the range $[-1 \cdot 10^9, 147483647]$; the end-point is $2^{31} - 1 - 2 \cdot 10^9 \simeq 1.47 \cdot 10^8$. Shifted values all have exactly 10 digits.

```
\int_const:Nn \c__fp_Bigg_leading_shift_int  { - 20 0000 }
\int_const:Nn \c__fp_Bigg_middle_shift_int   { 20 0000 *  9999 }
\int_const:Nn \c__fp_Bigg_trailing_shift_int { 20 0000 * 10000 }
\cs_new:Npn \__fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7;
  { + #1#2#3#4#5#6 ; {#7} }
```

(*End of definition for* `\__fp_pack_Bigg:NNNNNNw` *and others.*)

__fp_pack_twice_four:wNNNNNNNN

`\__fp_pack_twice_four:wNNNNNNNN` ⟨*tokens*⟩ `;` ⟨≥ 8 *digits*⟩

Grabs two sets of 4 digits and places them before the semi-colon delimiter. Putting several copies of this function before a semicolon packs more digits since each takes the digits packed by the others in its first argument.

```
\cs_new:Npn \__fp_pack_twice_four:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
  { #1 {#2#3#4#5} {#6#7#8#9} ; }
```

(*End of definition for* `\__fp_pack_twice_four:wNNNNNNNN`.)

__fp_pack_eight:wNNNNNNNN

`\__fp_pack_eight:wNNNNNNNN` ⟨*tokens*⟩ `;` ⟨≥ 8 *digits*⟩

Grabs one set of 8 digits and places them before the semi-colon delimiter as a single group. Putting several copies of this function before a semicolon packs more digits since each takes the digits packed by the others in its first argument.

```
\cs_new:Npn \__fp_pack_eight:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
  { #1 {#2#3#4#5#6#7#8#9} ; }
```

(*End of definition for* `\__fp_pack_eight:wNNNNNNNN`.)

__fp_basics_pack_low:NNNNNw
__fp_basics_pack_high:NNNNNw
__fp_basics_pack_high_carry:w

Addition and multiplication of significands are done in two steps: first compute a (more or less) exact result, then round and pack digits in the final (braced) form. These functions take care of the packing, with special attention given to the case where rounding has caused a carry. Since rounding can only shift the final digit by 1, a carry always produces an exact power of 10. Thus, `\__fp_basics_pack_high_carry:w` is always followed by four times `{0000}`.

This is used in l3fp-basics and l3fp-extended.

```
\cs_new:Npn \__fp_basics_pack_low:NNNNNw #1 #2#3#4#5 #6;
  { + #1 - 1 ; {#2#3#4#5} {#6} ; }
\cs_new:Npn \__fp_basics_pack_high:NNNNNw #1 #2#3#4#5 #6;
  {
    \if_meaning:w 2 #1
      \__fp_basics_pack_high_carry:w
    \fi:
    ; {#2#3#4#5} {#6}
  }
\cs_new:Npn \__fp_basics_pack_high_carry:w \fi: ; #1
  { \fi: + 1 ; {1000} }
```

(*End of definition for* `\__fp_basics_pack_low:NNNNNw`, `\__fp_basics_pack_high:NNNNNw`, *and* `\__fp_basics_pack_high_carry:w`.)

__fp_basics_pack_weird_low:NNNNw
__fp_basics_pack_weird_high:NNNNNNNNw

This is used in l3fp-basics for additions and divisions. Their syntax is confusing, hence the name.

```
\cs_new:Npn \__fp_basics_pack_weird_low:NNNNw #1 #2#3#4 #5;
  {
    \if_meaning:w 2 #1
      + 1
    \fi:
    \__fp_int_eval_end:
    #2#3#4; {#5} ;
  }
\cs_new:Npn \__fp_basics_pack_weird_high:NNNNNNNNw
  1 #1#2#3#4 #5#6#7#8 #9; { ; {#1#2#3#4} {#5#6#7#8} {#9} }
```

(*End of definition for* `\__fp_basics_pack_weird_low:NNNNw` *and* `\__fp_basics_pack_weird_high:NNNNNNNNw`.)

68.9 Decimate (dividing by a power of 10)

__fp_decimate:nNnnnn

`\__fp_decimate:nNnnnn {`⟨*shift*⟩`}` ⟨f_1⟩
`{`⟨X_1⟩`} {`⟨X_2⟩`} {`⟨X_3⟩`} {`⟨X_4⟩`}`

Each ⟨X_i⟩ consists in 4 digits exactly, and $1000 \le \langle X_1\rangle < 9999$. The first argument determines by how much we shift the digits. ⟨f_1⟩ is called as follows:

⟨f_1⟩ ⟨*rounding*⟩ `{`⟨X'_1⟩`} {`⟨X'_2⟩`}` ⟨*extra-digits*⟩ `;`

where $0 \le \langle X'_i\rangle < 10^8 - 1$ are 8 digit integers, forming the truncation of our number. In other words,

$$\left(\sum_{i=1}^{4} \langle X_i\rangle \cdot 10^{-4i} \cdot 10^{-\langle shift\rangle}\right) - \left(\langle X'_1\rangle \cdot 10^{-8} + \langle X'_2\rangle \cdot 10^{-16}\right) = 0.\langle extra\text{-}digits\rangle \cdot 10^{-16} \in [0, 10^{-16}).$$

To round properly later, we need to remember some information about the difference. The ⟨*rounding*⟩ digit is 0 if and only if the difference is exactly 0, and 5 if and only if the difference is exactly $0.5 \cdot 10^{-16}$. Otherwise, it is the (non-0, non-5) digit closest to 10^{17} times the difference. In particular, if the shift is 17 or more, all the digits are dropped, ⟨*rounding*⟩ is 1 (not 0), and ⟨X'_1⟩ and ⟨X'_2⟩ are both zero.

If the shift is 1, the ⟨*rounding*⟩ digit is simply the only digit that was pushed out of the brace groups (this is important for subtraction). It would be more natural for the ⟨*rounding*⟩ digit to be placed after the ⟨X'_i⟩, but the choice we make involves less reshuffling.

Note that this function treats negative ⟨*shift*⟩ as 0.

```
\cs_new:Npn \__fp_decimate:nNnnnn #1
  {
    \cs:w
      __fp_decimate_
      \if_int_compare:w \__fp_int_eval:w #1 > \c__fp_prec_int
        tiny
      \else:
        \__fp_int_to_roman:w \__fp_int_eval:w #1
      \fi:
      :Nnnnn
    \cs_end:
  }
```

Each of the auxiliaries see the function ⟨f_1⟩, followed by 4 blocks of 4 digits.

(*End of definition for* `\__fp_decimate:nNnnnn`.)

`\__fp_decimate_:Nnnnn`
`\__fp_decimate_tiny:Nnnnn`

If the ⟨*shift*⟩ is zero, or too big, life is very easy.

```
\cs_new:Npn \__fp_decimate_:Nnnnn #1 #2#3#4#5
  { #1 0 {#2#3} {#4#5} ; }
\cs_new:Npn \__fp_decimate_tiny:Nnnnn #1 #2#3#4#5
  { #1 1 { 0000 0000 } { 0000 0000 } 0 #2#3#4#5 ; }
```

(*End of definition for* `\__fp_decimate_:Nnnnn` *and* `\__fp_decimate_tiny:Nnnnn`.)

`\__fp_decimate_auxi:Nnnnn`
`\__fp_decimate_auxii:Nnnnn`
`\__fp_decimate_auxiii:Nnnnn`
`\__fp_decimate_auxiv:Nnnnn`
`\__fp_decimate_auxv:Nnnnn`
`\__fp_decimate_auxvi:Nnnnn`
`\__fp_decimate_auxvii:Nnnnn`
`\__fp_decimate_auxviii:Nnnnn`
`\__fp_decimate_auxix:Nnnnn`
`\__fp_decimate_auxx:Nnnnn`
`\__fp_decimate_auxxi:Nnnnn`
`\__fp_decimate_auxxii:Nnnnn`
`\__fp_decimate_auxxiii:Nnnnn`
`\__fp_decimate_auxxiv:Nnnnn`
`\__fp_decimate_auxxv:Nnnnn`
`\__fp_decimate_auxxvi:Nnnnn`

`\__fp_decimate_auxi:Nnnnn` ⟨f_1⟩ {⟨X_1⟩} {⟨X_2⟩} {⟨X_3⟩} {⟨X_4⟩}

Shifting happens in two steps: compute the ⟨*rounding*⟩ digit, and repack digits into two blocks of 8. The sixteen functions are very similar, and defined through `\__fp_tmp:w`. The arguments are as follows: `#1` indicates which function is being defined; after one step of expansion, `#2` yields the "extra digits" which are then converted by `\__fp_round_digit:Nw` to the ⟨*rounding*⟩ digit (note the `+` separating blocks of digits to avoid overflowing TEX's integers). This triggers the `f`-expansion of `\__fp_decimate_pack:nnnnnnnnnnw`,[10] responsible for building two blocks of 8 digits, and removing the rest. For this to work, `#3` alternates between braced and unbraced blocks of 4 digits, in such a way that the 5 first and 5 next token groups yield the correct blocks of 8 digits.

```
\cs_new:Npn \__fp_tmp:w #1 #2 #3
  {
    \cs_new:cpn { __fp_decimate_ #1 :Nnnnn } ##1 ##2##3##4##5
      {
        \exp_after:wN ##1
        \int_value:w
          \exp_after:wN \__fp_round_digit:Nw #2 ;
        \__fp_decimate_pack:nnnnnnnnnnw #3 ;
      }
  }
\__fp_tmp:w {i}   {\use_none:nnn      #50}{    0{#2}#3{#4}#5            }
\__fp_tmp:w {ii}  {\use_none:nn       #5 }{    00{#2}#3{#4}#5           }
\__fp_tmp:w {iii} {\use_none:n        #5 }{    000{#2}#3{#4}#5          }
\__fp_tmp:w {iv}  {                   #5 }{   {0000}#2{#3}#4 #5         }
\__fp_tmp:w {v}   {\use_none:nnn    #4#5 }{   0{0000}#2{#3}#4 #5        }
\__fp_tmp:w {vi}  {\use_none:nn     #4#5 }{   00{0000}#2{#3}#4 #5       }
\__fp_tmp:w {vii} {\use_none:n      #4#5 }{   000{0000}#2{#3}#4 #5      }
\__fp_tmp:w {viii}{                 #4#5 }{  {0000}0000{#2}#3 #4 #5     }
\__fp_tmp:w {ix}  {\use_none:nnn  #3#4+#5}{  0{0000}0000{#2}#3 #4 #5    }
\__fp_tmp:w {x}   {\use_none:nn   #3#4+#5}{  00{0000}0000{#2}#3 #4 #5   }
\__fp_tmp:w {xi}  {\use_none:n    #3#4+#5}{  000{0000}0000{#2}#3 #4 #5  }
\__fp_tmp:w {xii} {               #3#4+#5}{ {0000}0000{0000}#2 #3 #4 #5 }
\__fp_tmp:w {xiii}{\use_none:nnn#2#3+#4#5}{ 0{0000}0000{0000}#2 #3 #4 #5 }
\__fp_tmp:w {xiv} {\use_none:nn #2#3+#4#5}{ 00{0000}0000{0000}#2 #3 #4 #5 }
\__fp_tmp:w {xv}  {\use_none:n  #2#3+#4#5}{ 000{0000}0000{0000}#2 #3 #4 #5 }
\__fp_tmp:w {xvi} {             #2#3+#4#5}{{0000}0000{0000}0000 #2 #3 #4 #5}
```

(*End of definition for* `\__fp_decimate_auxi:Nnnnn` *and others*.)

[10] No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

__fp_decimate_pack:nnnnnnnnnnw The computation of the ⟨*rounding*⟩ digit leaves an unfinished `\int_value:w`, which expands the following functions. This allows us to repack nicely the digits we keep. Those digits come as an alternation of unbraced and braced blocks of 4 digits, such that the first 5 groups of token consist in 4 single digits, and one brace group (in some order), and the next 5 have the same structure. This is followed by some digits and a semicolon.

```
\cs_new:Npn \__fp_decimate_pack:nnnnnnnnnnw #1#2#3#4#5
  { \__fp_decimate_pack:nnnnnnw { #1#2#3#4#5 } }
\cs_new:Npn \__fp_decimate_pack:nnnnnnw #1 #2#3#4#5#6
  { {#1} {#2#3#4#5#6} }
```

(*End of definition for* `\__fp_decimate_pack:nnnnnnnnnnw`.)

68.10 Functions for use within primitive conditional branches

The functions described in this section are not pretty and can easily be misused. When correctly used, each of them removes one `\fi:` as part of its parameter text, and puts one back as part of its replacement text.

Many computation functions in l3fp must perform tests on the type of floating points that they receive. This is often done in an `\if_case:w` statement or another conditional statement, and only a few cases lead to actual computations: most of the special cases are treated using a few standard functions which we define now. A typical use context for those functions would be

```
\if_case:w ⟨integer⟩ \exp_stop_f:
     \__fp_case_return_o:Nw ⟨fp var⟩
\or: \__fp_case_use:nw {⟨some computation⟩}
\or: \__fp_case_return_same_o:w
\or: \__fp_case_return:nw {⟨something⟩}
\fi:
⟨junk⟩
⟨floating point⟩
```

In this example, the case 0 returns the floating point ⟨`fp var`⟩, expanding once after that floating point. Case 1 does ⟨`some computation`⟩ using the ⟨`floating point`⟩ (presumably compute the operation requested by the user in that non-trivial case). Case 2 returns the ⟨`floating point`⟩ without modifying it, removing the ⟨`junk`⟩ and expanding once after. Case 3 closes the conditional, removes the ⟨`junk`⟩ and the ⟨`floating point`⟩, and expands ⟨`something`⟩ next. In other cases, the "⟨`junk`⟩" is expanded, performing some other operation on the ⟨`floating point`⟩. We provide similar functions with two trailing ⟨`floating points`⟩.

__fp_case_use:nw This function ends a TeX conditional, removes junk until the next floating point, and places its first argument before that floating point, to perform some operation on the floating point.

```
\cs_new:Npn \__fp_case_use:nw #1#2 \fi: #3 \s__fp { \fi: #1 \s__fp }
```

(*End of definition for* `\__fp_case_use:nw`.)

`\__fp_case_return:nw` This function ends a TeX conditional, removes junk and a floating point, and places its first argument in the input stream. A quirk is that we don't define this function requiring a floating point to follow, simply anything ending in a semicolon. This, in turn, means that the ⟨*junk*⟩ may not contain semicolons.

```
\cs_new:Npn \__fp_case_return:nw #1#2 \fi: #3 ; { \fi: #1 }
```

(*End of definition for* `\__fp_case_return:nw`.)

`\__fp_case_return_o:Nw` This function ends a TeX conditional, removes junk and a floating point, and returns its first argument (an ⟨*fp var*⟩) then expands once after it.

```
\cs_new:Npn \__fp_case_return_o:Nw #1#2 \fi: #3 \s__fp #4 ;
  { \fi: \exp_after:wN #1 }
```

(*End of definition for* `\__fp_case_return_o:Nw`.)

`\__fp_case_return_same_o:w` This function ends a TeX conditional, removes junk, and returns the following floating point, expanding once after it.

```
\cs_new:Npn \__fp_case_return_same_o:w #1 \fi: #2 \s__fp
  { \fi: \__fp_exp_after_o:w \s__fp }
```

(*End of definition for* `\__fp_case_return_same_o:w`.)

`\__fp_case_return_o:Nww` Same as `\__fp_case_return_o:Nw` but with two trailing floating points.

```
\cs_new:Npn \__fp_case_return_o:Nww #1#2 \fi: #3 \s__fp #4 ; #5 ;
  { \fi: \exp_after:wN #1 }
```

(*End of definition for* `\__fp_case_return_o:Nww`.)

`\__fp_case_return_i_o:ww`
`\__fp_case_return_ii_o:ww` Similar to `\__fp_case_return_same_o:w`, but this returns the first or second of two trailing floating point numbers, expanding once after the result.

```
\cs_new:Npn \__fp_case_return_i_o:ww #1 \fi: #2 \s__fp #3 ; \s__fp #4 ;
  { \fi: \__fp_exp_after_o:w \s__fp #3 ; }
\cs_new:Npn \__fp_case_return_ii_o:ww #1 \fi: #2 \s__fp #3 ;
  { \fi: \__fp_exp_after_o:w }
```

(*End of definition for* `\__fp_case_return_i_o:ww` *and* `\__fp_case_return_ii_o:ww`.)

68.11 Integer floating points

`\__fp_int_p:w`
`\__fp_int:wTF` Tests if the floating point argument is an integer. For normal floating point numbers, this holds if the rounding digit resulting from `\__fp_decimate:nNnnnn` is 0.

```
\prg_new_conditional:Npnn \__fp_int:w \s__fp \__fp_chk:w #1 #2 #3 #4;
  { TF , T , F , p }
  {
    \if_case:w #1 \exp_stop_f:
          \prg_return_true:
    \or:
      \if_charcode:w 0
        \__fp_decimate:nNnnnn { \c__fp_prec_int - #3 }
          \__fp_use_i_until_s:nw #4
        \prg_return_true:
      \else:
        \prg_return_false:
```

```
      \fi:
    \else: \prg_return_false:
    \fi:
  }
```

(*End of definition for* `\__fp_int:wTF`.)

68.12 Small integer floating points

`\__fp_small_int:wTF`
`\__fp_small_int_true:wTF`
`\__fp_small_int_normal:NnwTF`
`\__fp_small_int_test:NnnwNTF`

Tests if the floating point argument is an integer or $\pm\infty$. If so, it is clipped to an integer in the range $[-10^8, 10^8]$ and fed as a braced argument to the ⟨*true code*⟩. Otherwise, the ⟨*false code*⟩ is performed.

First filter special cases: zeros and infinities are integers, `nan` is not. For normal numbers, decimate. If the rounding digit is not 0 run the ⟨*false code*⟩. If it is, then the integer is `#2` `#3`; use `#3` if `#2` vanishes and otherwise 10^8.

```
\cs_new:Npn \__fp_small_int:wTF \s__fp \__fp_chk:w #1#2
  {
    \if_case:w #1 \exp_stop_f:
          \__fp_case_return:nw { \__fp_small_int_true:wTF 0 ; }
    \or:   \exp_after:wN \__fp_small_int_normal:NnwTF
    \or:
      \__fp_case_return:nw
        {
          \exp_after:wN \__fp_small_int_true:wTF \int_value:w
            \if_meaning:w 2 #2 - \fi: 1 0000 0000 ;
        }
    \else: \__fp_case_return:nw \use_ii:nn
    \fi:
    #2
  }
\cs_new:Npn \__fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
\cs_new:Npn \__fp_small_int_normal:NnwTF #1#2#3;
  {
    \__fp_decimate:nNnnnn { \c__fp_prec_int - #2 }
      \__fp_small_int_test:NnnwNw
      #3 #1
  }
\cs_new:Npn \__fp_small_int_test:NnnwNw #1#2#3#4; #5
  {
    \if_meaning:w 0 #1
      \exp_after:wN \__fp_small_int_true:wTF
      \int_value:w \if_meaning:w 2 #5 - \fi:
        \if_int_compare:w #2 > \c_zero_int
          1 0000 0000
        \else:
          #3
        \fi:
      \exp_after:wN ;
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
  }
```

(*End of definition for* `\__fp_small_int:wTF` *and others.*)

68.13 Fast string comparison

__fp_str_if_eq:nn

A private version of the low-level string comparison function.

```
\cs_new_eq:NN \__fp_str_if_eq:nn \tex_strcmp:D
```

(*End of definition for* `\__fp_str_if_eq:nn`.)

68.14 Name of a function from its l3fp-parse name

__fp_func_to_name:N
__fp_func_to_name_aux:w

The goal is to convert for instance `\__fp_sin_o:w` to `sin`. This is used in error messages hence does not need to be fast.

```
\cs_new:Npn \__fp_func_to_name:N #1
  {
    \exp_last_unbraced:Nf
      \__fp_func_to_name_aux:w { \cs_to_str:N #1 } X
  }
\cs_set_protected:Npn \__fp_tmp:w #1 #2
  { \cs_new:Npn \__fp_func_to_name_aux:w ##1 #1 ##2 #2 ##3 X {##2} }
\exp_args:Nff \__fp_tmp:w { \tl_to_str:n { __fp_ } }
  { \tl_to_str:n { _o: } }
```

(*End of definition for* `\__fp_func_to_name:N` *and* `\__fp_func_to_name_aux:w`.)

68.15 Messages

Using a floating point directly is an error.

```
\msg_new:nnnn { fp } { misused }
  { A~floating~point~with~value~'#1'~was~misused. }
  {
    To~obtain~the~value~of~a~floating~point~variable,~use~
    '\token_to_str:N \fp_to_decimal:N',~
    '\token_to_str:N \fp_to_tl:N',~or~other~
    conversion~functions.
  }
\prop_gput:Nnn \g_msg_module_name_prop { fp } { LaTeX }
\prop_gput:Nnn \g_msg_module_type_prop { fp } { }
```

⟨/package⟩

Chapter 69

l3fp-traps implementation

```
⟨*package⟩
⟨@@=fp⟩
```

Exceptions should be accessed by an n-type argument, among

- invalid_operation
- division_by_zero
- overflow
- underflow
- inexact (actually never used).

69.1 Flags

\l_fp_invalid_operation_flag
\l_fp_division_by_zero_flag
\l_fp_overflow_flag
\l_fp_underflow_flag

Flags to denote exceptions.

```
\flag_new:N \l_fp_invalid_operation_flag
\flag_new:N \l_fp_division_by_zero_flag
\flag_new:N \l_fp_overflow_flag
\flag_new:N \l_fp_underflow_flag
```

(*End of definition for* \l_fp_invalid_operation_flag *and others. These variables are documented on page 269.*)

69.2 Traps

Exceptions can be trapped to obtain custom behaviour. When an invalid operation or a division by zero is trapped, the trap receives as arguments the result as an N-type floating point number, the function name (multiple letters for prefix operations, or a single symbol for infix operations), and the operand(s). When an overflow or underflow is trapped, the trap receives the resulting overly large or small floating point number if it is not too big, otherwise it receives $+\infty$. Currently, the inexact exception is entirely ignored.

The behaviour when an exception occurs is controlled by the definitions of the functions

- __fp_invalid_operation:nnw,

- __fp_invalid_operation_o:Nww,
- __fp_invalid_operation_tl_o:ff,
- __fp_division_by_zero_o:Nnw,
- __fp_division_by_zero_o:NNww,
- __fp_overflow:w,
- __fp_underflow:w.

Rather than changing them directly, we provide a user interface as \fp_trap:nn {⟨*exception*⟩} {⟨*way of trapping*⟩}, where the ⟨way of trapping⟩ is one of error, flag, or none.

We also provide __fp_invalid_operation_o:nw, defined in terms of __fp_invalid_operation:nnw.

\fp_trap:nn

```
\cs_new_protected:Npn \fp_trap:nn #1#2
  {
    \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
      {
        \clist_if_in:nnTF
          { invalid_operation , division_by_zero , overflow , underflow }
          {#1}
          {
            \msg_error:nnee { fp }
              { unknown-fpu-trap-type } {#1} {#2}
          }
          {
            \msg_error:nne
              { fp } { unknown-fpu-exception } {#1}
          }
      }
  }
```

(*End of definition for \fp_trap:nn. This function is documented on page 269.*)

__fp_trap_invalid_operation_set_error:
__fp_trap_invalid_operation_set_flag:
__fp_trap_invalid_operation_set_none:
__fp_trap_invalid_operation_set:N

We provide three types of trapping for invalid operations: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In most cases, the function produces as a result its first argument, possibly with post-expansion.

```
\cs_new_protected:Npn \__fp_trap_invalid_operation_set_error:
  { \__fp_trap_invalid_operation_set:N \prg_do_nothing: }
\cs_new_protected:Npn \__fp_trap_invalid_operation_set_flag:
  { \__fp_trap_invalid_operation_set:N \use_none:nnnnn }
\cs_new_protected:Npn \__fp_trap_invalid_operation_set_none:
  { \__fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
\cs_new_protected:Npn \__fp_trap_invalid_operation_set:N #1
  {
    \exp_args:Nno \use:n
      { \cs_set:Npn \__fp_invalid_operation:nnw ##1##2##3; }
      {
        #1
        \__fp_error:nnfn { invalid } {##2} { \fp_to_tl:n { ##3; } } { }
```

```
      \flag_ensure_raised:N \l_fp_invalid_operation_flag
      ##1
    }
  \exp_args:Nno \use:n
    { \cs_set:Npn \__fp_invalid_operation_o:Nww ##1##2; ##3; }
    {
      #1
      \__fp_error:nffn { invalid-ii }
        { \fp_to_tl:n { ##2; } } { \fp_to_tl:n { ##3; } } {##1}
      \flag_ensure_raised:N \l_fp_invalid_operation_flag
      \exp_after:wN \c_nan_fp
    }
  \exp_args:Nno \use:n
    { \cs_set:Npn \__fp_invalid_operation_tl_o:ff ##1##2 }
    {
      #1
      \__fp_error:nffn { invalid } {##1} {##2} { }
      \flag_ensure_raised:N \l_fp_invalid_operation_flag
      \exp_after:wN \c_nan_fp
    }
}
```

(*End of definition for* `\__fp_trap_invalid_operation_set_error:` *and others.*)

__fp_trap_division_by_zero_set_error:
__fp_trap_division_by_zero_set_flag:
__fp_trap_division_by_zero_set_none:
__fp_trap_division_by_zero_set:N

We provide three types of trapping for invalid operations and division by zero: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In all cases, the function must produce a result, namely its first argument, $\pm\infty$ or `nan`.

```
\cs_new_protected:Npn \__fp_trap_division_by_zero_set_error:
  { \__fp_trap_division_by_zero_set:N \prg_do_nothing: }
\cs_new_protected:Npn \__fp_trap_division_by_zero_set_flag:
  { \__fp_trap_division_by_zero_set:N \use_none:nnnnn }
\cs_new_protected:Npn \__fp_trap_division_by_zero_set_none:
  { \__fp_trap_division_by_zero_set:N \use_none:nnnnnnn }
\cs_new_protected:Npn \__fp_trap_division_by_zero_set:N #1
  {
    \exp_args:Nno \use:n
      { \cs_set:Npn \__fp_division_by_zero_o:Nnw ##1##2##3; }
      {
        #1
        \__fp_error:nnfn { zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
        \flag_ensure_raised:N \l_fp_division_by_zero_flag
        \exp_after:wN ##1
      }
    \exp_args:Nno \use:n
      { \cs_set:Npn \__fp_division_by_zero_o:NNww ##1##2##3; ##4; }
      {
        #1
        \__fp_error:nffn { zero-div-ii }
          { \fp_to_tl:n { ##3; } } { \fp_to_tl:n { ##4; } } {##2}
        \flag_ensure_raised:N \l_fp_division_by_zero_flag
        \exp_after:wN ##1
      }
  }
```

(*End of definition for* `\__fp_trap_division_by_zero_set_error:` *and others.*)

`\__fp_trap_overflow_set_error:`
`\__fp_trap_overflow_set_flag:`
`\__fp_trap_overflow_set_none:`
`\__fp_trap_overflow_set:N`
`\__fp_trap_underflow_set_error:`
`\__fp_trap_underflow_set_flag:`
`\__fp_trap_underflow_set_none:`
`\__fp_trap_underflow_set:N`
`\__fp_trap_overflow_set:NnNn`

Just as for invalid operations and division by zero, the three different behaviours are obtained by feeding `\prg_do_nothing:`, `\use_none:nnnnn` or `\use_none:nnnnnnn` to an auxiliary, with a further auxiliary common to overflow and underflow functions. In most cases, the argument of the `\__fp_overflow:w` and `\__fp_underflow:w` functions will be an (almost) normal number (with an exponent outside the allowed range), and the error message thus displays that number together with the result to which it overflowed or underflowed. For extreme cases such as `10 ** 1e9999`, the exponent would be too large for TeX, and `\__fp_overflow:w` receives $\pm\infty$ (`\__fp_underflow:w` would receive ± 0); then we cannot do better than simply say an overflow or underflow occurred.

```
\cs_new_protected:Npn \__fp_trap_overflow_set_error:
  { \__fp_trap_overflow_set:N \prg_do_nothing: }
\cs_new_protected:Npn \__fp_trap_overflow_set_flag:
  { \__fp_trap_overflow_set:N \use_none:nnnnn }
\cs_new_protected:Npn \__fp_trap_overflow_set_none:
  { \__fp_trap_overflow_set:N \use_none:nnnnnnn }
\cs_new_protected:Npn \__fp_trap_overflow_set:N #1
  { \__fp_trap_overflow_set:NnNn #1 { overflow } \__fp_inf_fp:N { inf } }
\cs_new_protected:Npn \__fp_trap_underflow_set_error:
  { \__fp_trap_underflow_set:N \prg_do_nothing: }
\cs_new_protected:Npn \__fp_trap_underflow_set_flag:
  { \__fp_trap_underflow_set:N \use_none:nnnnn }
\cs_new_protected:Npn \__fp_trap_underflow_set_none:
  { \__fp_trap_underflow_set:N \use_none:nnnnnnn }
\cs_new_protected:Npn \__fp_trap_underflow_set:N #1
  { \__fp_trap_overflow_set:NnNn #1 { underflow } \__fp_zero_fp:N { 0 } }
\cs_new_protected:Npn \__fp_trap_overflow_set:NnNn #1#2#3#4
  {
    \exp_args:Nno \use:n
      { \cs_set:cpn { __fp_ #2 :w } \s__fp \__fp_chk:w ##1##2##3; }
      {
        #1
        \__fp_error:nffn
          { flow \if_meaning:w 1 ##1 -to \fi: }
          { \fp_to_tl:n { \s__fp \__fp_chk:w ##1##2##3; } }
          { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }
          {#2}
        \flag_ensure_raised:c { l_fp_#2_flag }
        #3 ##2
      }
  }
```

(*End of definition for* `\__fp_trap_overflow_set_error:` *and others.*)

`\__fp_invalid_operation:nnw`
`\__fp_invalid_operation_o:Nww`
`\__fp_invalid_operation_tl_o:ff`
`\__fp_division_by_zero_o:Nnw`
`\__fp_division_by_zero_o:NNww`
`\__fp_overflow:w`
`\__fp_underflow:w`

Initialize the control sequences (to log properly their existence). Then set invalid operations to trigger an error, and division by zero, overflow, and underflow to act silently on their flag.

```
\cs_new:Npn \__fp_invalid_operation:nnw #1#2#3; { }
\cs_new:Npn \__fp_invalid_operation_o:Nww #1#2; #3; { }
\cs_new:Npn \__fp_invalid_operation_tl_o:ff #1 #2 { }
\cs_new:Npn \__fp_division_by_zero_o:Nnw #1#2#3; { }
\cs_new:Npn \__fp_division_by_zero_o:NNww #1#2#3; #4; { }
```

```
\cs_new:Npn \__fp_overflow:w { }
\cs_new:Npn \__fp_underflow:w { }
\fp_trap:nn { invalid_operation } { error }
\fp_trap:nn { division_by_zero } { flag }
\fp_trap:nn { overflow } { flag }
\fp_trap:nn { underflow } { flag }
```

(*End of definition for* `\__fp_invalid_operation:nnw` *and others.*)

`\__fp_invalid_operation_o:nw`
`\__fp_invalid_operation_o:fw`

Convenient short-hands for returning `\c_nan_fp` for a unary or binary operation, and expanding after.

```
\cs_new:Npn \__fp_invalid_operation_o:nw
  { \__fp_invalid_operation:nnw { \exp_after:wN \c_nan_fp } }
\cs_generate_variant:Nn \__fp_invalid_operation_o:nw { f }
```

(*End of definition for* `\__fp_invalid_operation_o:nw`.)

69.3 Errors

`\__fp_error:nnnn`
`\__fp_error:nnfn`
`\__fp_error:nffn`
`\__fp_error:nfff`

```
\cs_new:Npn \__fp_error:nnnn
  { \msg_expandable_error:nnnnn { fp } }
\cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff , nfff }
```

(*End of definition for* `\__fp_error:nnnn`.)

69.4 Messages

Some messages.

```
\msg_new:nnnn { fp } { unknown-fpu-exception }
  {
    The~FPU~exception~'#1'~is~not~known:~
    that~trap~will~never~be~triggered.
  }
  {
    The~only~exceptions~to~which~traps~can~be~attached~are \\
    \iow_indent:n
      {
        * ~ invalid_operation \\
        * ~ division_by_zero \\
        * ~ overflow \\
        * ~ underflow
      }
  }
\msg_new:nnnn { fp } { unknown-fpu-trap-type }
  { The~FPU~trap~type~'#2'~is~not~known. }
  {
    The~trap~type~must~be~one~of \\
    \iow_indent:n
      {
        * ~ error \\
        * ~ flag \\
```

```
        * ~ none
      }
  }
\msg_new:nnn { fp } { flow }
  { An ~ #3 ~ occurred. }
\msg_new:nnn { fp } { flow-to }
  { #1 ~ #3 ed ~ to ~ #2 . }
\msg_new:nnn { fp } { zero-div }
  { Division~by~zero~in~ #1 (#2) }
\msg_new:nnn { fp } { zero-div-ii }
  { Division~by~zero~in~ (#1) #3 (#2) }
\msg_new:nnn { fp } { invalid }
  { Invalid~operation~ #1 (#2) }
\msg_new:nnn { fp } { invalid-ii }
  { Invalid~operation~ (#1) #3 (#2) }
\msg_new:nnn { fp } { unknown-type }
  { Unknown~type~for~'#1' }
```

⟨/package⟩

Chapter 70

l3fp-round implementation

```
⟨*package⟩
⟨@@=fp⟩
```

__fp_parse_word_trunc:N
__fp_parse_word_floor:N
__fp_parse_word_ceil:N

```
\cs_new:Npn \__fp_parse_word_trunc:N
  { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_zero:NNN }
\cs_new:Npn \__fp_parse_word_floor:N
  { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_ninf:NNN }
\cs_new:Npn \__fp_parse_word_ceil:N
  { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_pinf:NNN }
```

(*End of definition for* `\__fp_parse_word_trunc:N`*,* `\__fp_parse_word_floor:N`*, and* `\__fp_parse_word_ceil:N`*.*)

__fp_parse_word_round:N
__fp_parse_round:Nw

```
\cs_new:Npn \__fp_parse_word_round:N #1#2
  {
    \__fp_parse_function:NNN
      \__fp_round_o:Nw \__fp_round_to_nearest:NNN #1
    #2
  }
\cs_new:Npn \__fp_parse_round:Nw #1 #2 \__fp_round_to_nearest:NNN #3#4
  { #2 #1 #3 }

```

(*End of definition for* `\__fp_parse_word_round:N` *and* `\__fp_parse_round:Nw`*.*)

70.1 Rounding tools

\c__fp_five_int This is used as the half-point for which numbers are rounded up/down.

```
\int_const:Nn \c__fp_five_int { 5 }
```

(*End of definition for* `\c__fp_five_int`*.*)

Floating point operations often yield a result that cannot be exactly represented in a significand with 16 digits. In that case, we need to round the exact result to a representable number. The IEEE standard defines four rounding modes:

- Round to nearest: round to the representable floating point number whose absolute difference with the exact result is the smallest. If the exact result lies exactly at the mid-point between two consecutive representable floating point numbers, round to the floating point number whose last digit is even.
- Round towards negative infinity: round to the greatest floating point number not larger than the exact result.
- Round towards zero: round to a floating point number with the same sign as the exact result, with the largest absolute value not larger than the absolute value of the exact result.
- Round towards positive infinity: round to the least floating point number not smaller than the exact result.

This is not fully implemented in l3fp yet, and transcendental functions fall back on the "round to nearest" mode. All rounding for basic algebra is done through the functions defined in this module, which can be redefined to change their rounding behaviour (but there is not interface for that yet).

The rounding tools available in this module are many variations on a base function `\__fp_round:NNN`, which expands to `0\exp_stop_f:` or `1\exp_stop_f:` depending on whether the final result should be rounded up or down.

- `\__fp_round:NNN` ⟨*sign*⟩ ⟨*digit*$_1$⟩ ⟨*digit*$_2$⟩ can expand to `0\exp_stop_f:` or `1\exp_stop_f:`.
- `\__fp_round_s:NNNw` ⟨*sign*⟩ ⟨*digit*$_1$⟩ ⟨*digit*$_2$⟩ ⟨*more digits*⟩`;` can expand to `0\exp_stop_f:;` or `1\exp_stop_f:;`.
- `\__fp_round_neg:NNN` ⟨*sign*⟩ ⟨*digit*$_1$⟩ ⟨*digit*$_2$⟩ can expand to `0\exp_stop_f:` or `1\exp_stop_f:`.

See implementation comments for details on the syntax.

`\__fp_round:NNN`
`\__fp_round_to_nearest:NNN`
`\__fp_round_to_nearest_ninf:NNN`
`\__fp_round_to_nearest_zero:NNN`
`\__fp_round_to_nearest_pinf:NNN`
`\__fp_round_to_ninf:NNN`
`\__fp_round_to_zero:NNN`
`\__fp_round_to_pinf:NNN`

`\__fp_round:NNN` ⟨*final sign*⟩ ⟨*digit*$_1$⟩ ⟨*digit*$_2$⟩

If rounding the number ⟨*final sign*⟩⟨*digit*$_1$⟩.⟨*digit*$_2$⟩ to an integer rounds it towards zero (truncates it), this function expands to `0\exp_stop_f:`, and otherwise to `1\exp_stop_f:`. Typically used within the scope of an `\__fp_int_eval:w`, to add 1 if needed, and thereby round correctly. The result depends on the rounding mode.

It is very important that ⟨*final sign*⟩ be the final sign of the result. Otherwise, the result would be incorrect in the case of rounding towards $-\infty$ or towards $+\infty$. Also recall that ⟨*final sign*⟩ is 0 for positive, and 2 for negative.

By default, the functions below return `0\exp_stop_f:`, but this is superseded by `\__fp_round_return_one:`, which instead returns `1\exp_stop_f:`, expanding everything and removing `0\exp_stop_f:` in the process. In the case of rounding towards $\pm\infty$ or towards 0, this is not really useful, but it prepares us for the "round to nearest, ties to even" mode.

The "round to nearest" mode is the default. If the ⟨*digit*$_2$⟩ is larger than 5, then round up. If it is less than 5, round down. If it is exactly 5, then round such that ⟨*digit*$_1$⟩ plus the result is even. In other words, round up if ⟨*digit*$_1$⟩ is odd.

The "round to nearest" mode has three variants, which differ in how ties are rounded: down towards $-\infty$, truncated towards 0, or up towards $+\infty$.

```
\cs_new:Npn \__fp_round_return_one:
  { \exp_after:wN 1 \exp_after:wN \exp_stop_f: \exp:w }
\cs_new:Npn \__fp_round_to_ninf:NNN #1 #2 #3
  {
    \if_meaning:w 2 #1
      \if_int_compare:w #3 > \c_zero_int
        \__fp_round_return_one:
      \fi:
    \fi:
    \c_zero_int
  }
\cs_new:Npn \__fp_round_to_zero:NNN #1 #2 #3 { \c_zero_int }
\cs_new:Npn \__fp_round_to_pinf:NNN #1 #2 #3
  {
    \if_meaning:w 0 #1
      \if_int_compare:w #3 > \c_zero_int
        \__fp_round_return_one:
      \fi:
    \fi:
    \c_zero_int
  }
\cs_new:Npn \__fp_round_to_nearest:NNN #1 #2 #3
  {
    \if_int_compare:w #3 > \c__fp_five_int
      \__fp_round_return_one:
    \else:
      \if_meaning:w 5 #3
        \if_int_odd:w #2 \exp_stop_f:
          \__fp_round_return_one:
        \fi:
      \fi:
    \fi:
    \c_zero_int
  }
\cs_new:Npn \__fp_round_to_nearest_ninf:NNN #1 #2 #3
  {
    \if_int_compare:w #3 > \c__fp_five_int
      \__fp_round_return_one:
    \else:
      \if_meaning:w 5 #3
        \if_meaning:w 2 #1
            \__fp_round_return_one:
        \fi:
      \fi:
    \fi:
    \c_zero_int
  }
\cs_new:Npn \__fp_round_to_nearest_zero:NNN #1 #2 #3
  {
    \if_int_compare:w #3 > \c__fp_five_int
      \__fp_round_return_one:
    \fi:
    \c_zero_int
  }
```

```
\cs_new:Npn \__fp_round_to_nearest_pinf:NNN #1 #2 #3
  {
    \if_int_compare:w #3 > \c__fp_five_int
      \__fp_round_return_one:
    \else:
      \if_meaning:w 5 #3
        \if_meaning:w 0 #1
            \__fp_round_return_one:
        \fi:
      \fi:
    \fi:
    \c_zero_int
  }
\cs_new_eq:NN \__fp_round:NNN \__fp_round_to_nearest:NNN
```

(*End of definition for* `\__fp_round:NNN` *and others.*)

`\__fp_round_s:NNNw`

`\__fp_round_s:NNNw` ⟨*final sign*⟩ ⟨*digit*⟩ ⟨*more digits*⟩ `;`

Similar to `\__fp_round:NNN`, but with an extra semicolon, this function expands to `0\exp_stop_f:;` if rounding ⟨*final sign*⟩⟨*digit*⟩.⟨*more digits*⟩ to an integer truncates, and to `1\exp_stop_f:;` otherwise. The ⟨*more digits*⟩ part must be a digit, followed by something that does not overflow a `\int_use:N \__fp_int_eval:w` construction. The only relevant information about this piece is whether it is zero or not.

```
\cs_new:Npn \__fp_round_s:NNNw #1 #2 #3 #4;
  {
    \exp_after:wN \__fp_round:NNN
    \exp_after:wN #1
    \exp_after:wN #2
    \int_value:w \__fp_int_eval:w
      \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
                      \if_meaning:w 5 #3 1 \fi:
                \exp_stop_f:
        \if_int_compare:w \__fp_int_eval:w #4 > \c_zero_int
          1 +
        \fi:
      \fi:
      #3
    ;
  }
```

(*End of definition for* `\__fp_round_s:NNNw`.)

`\__fp_round_digit:Nw`

`\int_value:w \__fp_round_digit:Nw` ⟨*digit*⟩ ⟨*int expr*⟩ `;`

This function should always be called within an `\int_value:w` or `\__fp_int_eval:w` expansion; it may add an extra `\__fp_int_eval:w`, which means that the integer or integer expression should not be ended with a synonym of `\relax`, but with a semi-colon for instance.

```
\cs_new:Npn \__fp_round_digit:Nw #1 #2;
  {
    \if_int_odd:w \if_meaning:w 0 #1 1 \else:
                  \if_meaning:w 5 #1 1 \else:
                  0 \fi: \fi: \exp_stop_f:
      \if_int_compare:w \__fp_int_eval:w #2 > \c_zero_int
        \__fp_int_eval:w 1 +
```

```
      \fi:
    \fi:
    #1
  }
```

(*End of definition for* `\__fp_round_digit:Nw`.)

`\__fp_round_neg:NNN`
`\__fp_round_to_nearest_neg:NNN`
`\__fp_round_to_nearest_ninf_neg:NNN`
`\__fp_round_to_nearest_zero_neg:NNN`
`\__fp_round_to_nearest_pinf_neg:NNN`
`\__fp_round_to_ninf_neg:NNN`
`\__fp_round_to_zero_neg:NNN`
`\__fp_round_to_pinf_neg:NNN`

`\__fp_round_neg:NNN` ⟨*final sign*⟩ ⟨*digit*$_1$⟩ ⟨*digit*$_2$⟩

This expands to `0\exp_stop_f:` or `1\exp_stop_f:` after doing the following test. Starting from a number of the form ⟨*final sign*⟩0.⟨*15 digits*⟩⟨*digit*$_1$⟩ with exactly 15 (non-all-zero) digits before ⟨*digit*$_1$⟩, subtract from it ⟨*final sign*⟩0.0 . . . 0⟨*digit*$_2$⟩, where there are 16 zeros. If in the current rounding mode the result should be rounded down, then this function returns `1\exp_stop_f:`. Otherwise, *i.e.*, if the result is rounded back to the first operand, then this function returns `0\exp_stop_f:`.

It turns out that this negative "round to nearest" is identical to the positive one. And this is the default mode.

```
\cs_new_eq:NN \__fp_round_to_ninf_neg:NNN \__fp_round_to_pinf:NNN
\cs_new:Npn \__fp_round_to_zero_neg:NNN #1 #2 #3
  {
    \if_int_compare:w #3 > \c_zero_int
      \__fp_round_return_one:
    \fi:
    \c_zero_int
  }
\cs_new_eq:NN \__fp_round_to_pinf_neg:NNN \__fp_round_to_ninf:NNN
\cs_new_eq:NN \__fp_round_to_nearest_neg:NNN \__fp_round_to_nearest:NNN
\cs_new_eq:NN \__fp_round_to_nearest_ninf_neg:NNN
  \__fp_round_to_nearest_pinf:NNN
\cs_new:Npn \__fp_round_to_nearest_zero_neg:NNN #1 #2 #3
  {
    \if_int_compare:w #3 < \c__fp_five_int \else:
      \__fp_round_return_one:
    \fi:
    \c_zero_int
  }
\cs_new_eq:NN \__fp_round_to_nearest_pinf_neg:NNN
  \__fp_round_to_nearest_ninf:NNN
\cs_new_eq:NN \__fp_round_neg:NNN \__fp_round_to_nearest_neg:NNN
```

(*End of definition for* `\__fp_round_neg:NNN` *and others.*)

70.2 The round function

`\__fp_round_o:Nw`
`\__fp_round_aux_o:Nw`

First check that all arguments are floating point numbers. The `trunc`, `ceil` and `floor` functions expect one or two arguments (the second is 0 by default), and the `round` function also accepts a third argument (`nan` by default), which changes `#1` from `\__fp_round_to_nearest:NNN` to one of its analogues.

```
\cs_new:Npn \__fp_round_o:Nw #1
  {
    \__fp_parse_function_all_fp_o:fnw
      { \__fp_round_name_from_cs:N #1 }
      { \__fp_round_aux_o:Nw #1 }
```

```
  }
\cs_new:Npn \__fp_round_aux_o:Nw #1#2 @
  {
    \if_case:w
      \__fp_int_eval:w \__fp_array_count:n {#2} \__fp_int_eval_end:
         \__fp_round_no_arg_o:Nw #1 \exp:w
    \or: \__fp_round:Nwn #1 #2 {0} \exp:w
    \or: \__fp_round:Nww #1 #2 \exp:w
    \else: \__fp_round:Nwww #1 #2 @ \exp:w
    \fi:
    \exp_after:wN \exp_end:
  }
```

(*End of definition for* `\__fp_round_o:Nw` *and* `\__fp_round_aux_o:Nw`.)

`\__fp_round_no_arg_o:Nw`

```
\cs_new:Npn \__fp_round_no_arg_o:Nw #1
  {
    \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
      { \__fp_error:nnnn { num-args } { round () } { 1 } { 3 } }
      {
        \__fp_error:nffn { num-args }
          { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
      }
    \exp_after:wN \c_nan_fp
  }
```

(*End of definition for* `\__fp_round_no_arg_o:Nw`.)

`\__fp_round:Nwww` Having three arguments is only allowed for `round`, not `trunc`, `ceil`, `floor`, so check for that case. If all is well, construct one of `\__fp_round_to_nearest:NNN`, `\__fp_round_to_nearest_zero:NNN`, `\__fp_round_to_nearest_ninf:NNN`, `\__fp_round_to_nearest_pinf:NNN` and act accordingly.

```
\cs_new:Npn \__fp_round:Nwww #1#2 ; #3 ; \s__fp \__fp_chk:w #4#5#6 ; #7 @
  {
    \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
      {
        \tl_if_empty:nTF {#7}
          {
            \exp_args:Nc \__fp_round:Nww
              {
                __fp_round_to_nearest
                \if_meaning:w 0 #4 _zero \else:
                \if_case:w #5 \exp_stop_f: _pinf \or: \else: _ninf \fi: \fi:
                :NNN
              }
            #2 ; #3 ;
          }
          {
            \__fp_error:nnnn { num-args } { round () } { 1 } { 3 }
            \exp_after:wN \c_nan_fp
          }
      }
      {
```

```
      \__fp_error:nffn { num-args }
        { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
      \exp_after:wN \c_nan_fp
    }
}
```

(*End of definition for* `\__fp_round:Nwww`.)

`\__fp_round_name_from_cs:N`

```
\cs_new:Npn \__fp_round_name_from_cs:N #1
  {
    \cs_if_eq:NNTF #1 \__fp_round_to_zero:NNN { trunc }
      {
        \cs_if_eq:NNTF #1 \__fp_round_to_ninf:NNN { floor }
          {
            \cs_if_eq:NNTF #1 \__fp_round_to_pinf:NNN { ceil }
              { round }
          }
      }
  }
```

(*End of definition for* `\__fp_round_name_from_cs:N`.)

`\__fp_round:Nww`
`\__fp_round:Nwn`
`\__fp_round_normal:NwNNnw`
`\__fp_round_normal:NnnwNNnn`
`\__fp_round_pack:Nw`
`\__fp_round_normal:NNwNnn`
`\__fp_round_normal_end:wwNnn`
`\__fp_round_special:NwwNnn`
`\__fp_round_special_aux:Nw`

If the number of digits to round to is an integer or infinity all is good; if it is `nan` then just produce a `nan`; otherwise invalid as we have something like `round(1,3.14)` where the number of digits is not an integer.

```
\cs_new:Npn \__fp_round:Nww #1#2 ; #3 ;
  {
    \__fp_small_int:wTF #3; { \__fp_round:Nwn #1#2; }
      {
        \if:w 3 \__fp_kind:w #3 ;
          \exp_after:wN \use_i:nn
        \else:
          \exp_after:wN \use_ii:nn
        \fi:
        { \exp_after:wN \c_nan_fp }
        {
          \__fp_invalid_operation_tl_o:ff
            { \__fp_round_name_from_cs:N #1 }
            { \__fp_array_to_clist:n { #2; #3; } }
        }
      }
  }
\cs_new:Npn \__fp_round:Nwn #1 \s__fp \__fp_chk:w #2#3#4; #5
  {
    \if_meaning:w 1 #2
      \exp_after:wN \__fp_round_normal:NwNNnw
      \exp_after:wN #1
      \int_value:w #5
    \else:
      \exp_after:wN \__fp_exp_after_o:w
    \fi:
    \s__fp \__fp_chk:w #2#3#4;
  }
```

```
\cs_new:Npn \__fp_round_normal:NwNNnw #1#2 \s__fp \__fp_chk:w 1#3#4#5;
  {
    \__fp_decimate:nNnnnn { \c__fp_prec_int - #4 - #2 }
      \__fp_round_normal:NnnwNNnn #5 #1 #3 {#4} {#2}
  }
\cs_new:Npn \__fp_round_normal:NnnwNNnn #1#2#3#4; #5#6
  {
    \exp_after:wN \__fp_round_normal:NNwNnn
    \int_value:w \__fp_int_eval:w
      \if_int_compare:w #2 > \c_zero_int
        1 \int_value:w #2
        \exp_after:wN \__fp_round_pack:Nw
        \int_value:w \__fp_int_eval:w 1#3 +
      \else:
        \if_int_compare:w #3 > \c_zero_int
          1 \int_value:w #3 +
        \fi:
      \fi:
      \exp_after:wN #5
      \exp_after:wN #6
      \use_none:nnnnnnn #3
      #1
      \__fp_int_eval_end:
      0000 0000 0000 0000 ; #6
  }
\cs_new:Npn \__fp_round_pack:Nw #1
  { \if_meaning:w 2 #1 + 1 \fi: \__fp_int_eval_end: }
\cs_new:Npn \__fp_round_normal:NNwNnn #1 #2
  {
    \if_meaning:w 0 #2
      \exp_after:wN \__fp_round_special:NwwNnn
      \exp_after:wN #1
    \fi:
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_round_normal_end:wwNnn
    ; #2
  }
\cs_new:Npn \__fp_round_normal_end:wwNnn #1;#2;#3#4#5
  {
    \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
    \__fp_sanitize:Nw #3 #4 ; #1 ;
  }
\cs_new:Npn \__fp_round_special:NwwNnn #1#2;#3;#4#5#6
  {
    \if_meaning:w 0 #1
      \__fp_case_return:nw
        { \exp_after:wN \__fp_zero_fp:N \exp_after:wN #4 }
    \else:
      \exp_after:wN \__fp_round_special_aux:Nw
      \exp_after:wN #4
      \int_value:w \__fp_int_eval:w 1
        \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
    \fi:
```

```
    ;
  }
\cs_new:Npn \__fp_round_special_aux:Nw #1#2;
  {
    \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
    \__fp_sanitize:Nw #1#2; {1000}{0000}{0000}{0000};
  }
```

(*End of definition for* `\__fp_round:Nww` *and others.*)

```
⟨/package⟩
```

Chapter 71

l3fp-parse implementation

24074 ⟨*package⟩

24075 ⟨@@=fp⟩

71.1 Work plan

The task at hand is non-trivial, and some previous failed attempts show that the code leads to unreadable logs, so we had better get it (almost) right the first time. Let us first describe our goal, then discuss the design precisely before writing any code.

In this file at least, a ⟨*floating point object*⟩ is a floating point number or tuple. This can be extended to anything that starts with `\s__fp` or `\s__fp_`⟨*type*⟩ and ends with `;` with some internal structure that depends on the ⟨*type*⟩.

`\__fp_parse:n`

`\__fp_parse:n {`⟨*fp expr*⟩`}`

Evaluates the ⟨*fp expr*⟩ and leaves the result in the input stream as a floating point object. This function forms the basis of almost all public l3fp functions. During evaluation, each token is fully `f`-expanded.

`\__fp_parse_o:n` does the same but expands once after its result.

TEXhackers note: Registers (integers, toks, etc.) are automatically unpacked, without requiring a function such as `\int_use:N`. Invalid tokens remaining after `f`-expansion lead to unrecoverable low-level TEX errors.

(*End of definition for* `\__fp_parse:n`.)

`\c__fp_prec_func_int`
`\c__fp_prec_hatii_int`
`\c__fp_prec_hat_int`
`\c__fp_prec_not_int`
`\c__fp_prec_juxt_int`
`\c__fp_prec_times_int`
`\c__fp_prec_plus_int`
`\c__fp_prec_comp_int`
`\c__fp_prec_and_int`
`\c__fp_prec_or_int`
`\c__fp_prec_quest_int`
`\c__fp_prec_colon_int`
`\c__fp_prec_comma_int`
`\c__fp_prec_tuple_int`
`\c__fp_prec_end_int`

Floating point expressions are composed of numbers, given in various forms, infix operators, such as `+`, `**`, or `,` (which joins two numbers into a list), and prefix operators, such as the unary `-`, functions, or opening parentheses. Here is a list of precedences which control the order of evaluation (some distinctions are irrelevant for the order of evaluation, but serve as signals), from the tightest binding to the loosest binding.

16 Function calls.

13/14 Binary `**` and `^` (right to left).

12 Unary `+`, `-`, `!` (right to left).

11 Juxtaposition (implicit `*`) with no parenthesis.

10 Binary `*` and `/`.

9 Binary `+` and `-`.

7 Comparisons.

6 Logical `and`, denoted by `&&`.

5 Logical `or`, denoted by `||`.

4 Ternary operator `?:`, piece `?`.

3 Ternary operator `?:`, piece `:`.

2 Commas.

1 Place where a comma is allowed and generates a tuple.

0 Start and end of the expression.

```
\int_const:Nn \c__fp_prec_func_int   { 16 }
\int_const:Nn \c__fp_prec_hatii_int  { 14 }
\int_const:Nn \c__fp_prec_hat_int    { 13 }
\int_const:Nn \c__fp_prec_not_int    { 12 }
\int_const:Nn \c__fp_prec_juxt_int   { 11 }
\int_const:Nn \c__fp_prec_times_int  { 10 }
\int_const:Nn \c__fp_prec_plus_int   { 9 }
\int_const:Nn \c__fp_prec_comp_int   { 7 }
\int_const:Nn \c__fp_prec_and_int    { 6 }
\int_const:Nn \c__fp_prec_or_int     { 5 }
\int_const:Nn \c__fp_prec_quest_int  { 4 }
\int_const:Nn \c__fp_prec_colon_int  { 3 }
\int_const:Nn \c__fp_prec_comma_int  { 2 }
\int_const:Nn \c__fp_prec_tuple_int  { 1 }
\int_const:Nn \c__fp_prec_end_int    { 0 }
```

(*End of definition for* `\c__fp_prec_func_int` *and others.*)

71.1.1 Storing results

The main question in parsing expressions expandably is to decide where to put the intermediate results computed for various subexpressions.

One option is to store the values at the start of the expression, and carry them together as the first argument of each macro. However, we want to `f`-expand tokens one by one in the expression (as `\int_eval:n` does), and with this approach, expanding the next unread token forces us to jump with `\exp_after:wN` over every value computed earlier in the expression. With this approach, the run-time grows at least quadratically in the length of the expression, if not as its cube (inserting the `\exp_after:wN` is tricky and slow).

A second option is to place those values at the end of the expression. Then expanding the next unread token is straightforward, but this still hits a performance issue: for long expressions we would be reaching all the way to the end of the expression at every step of the calculation. The run-time is again quadratic.

A variation of the above attempts to place the intermediate results which appear when computing a parenthesized expression near the closing parenthesis. This still lets us expand tokens as we go, and avoids performance problems as long as there are enough parentheses. However, it would be better to avoid requiring the closing parenthesis to be present as soon as the corresponding opening parenthesis is read: the closing parenthesis may still be hidden in a macro yet to be expanded.

Hence, we need to go for some fine expansion control: the result is stored *before* the start!

Let us illustrate this idea in a simple model: adding positive integers which may be resulting from the expansion of macros, or may be values of registers. Assume that one number, say, 12345, has already been found, and that we want to parse the next number. The current status of the code may look as follows.

```
\exp_after:wN \add:ww \int_value:w 12345 \exp_after:wN ;
\exp:w \operand:w ⟨stuff⟩
```

One step of expansion expands `\exp_after:wN`, which triggers the primitive `\int_-value:w`, which reads the five digits we have already found, `12345`. This integer is unfinished, causing the second `\exp_after:wN` to expand, and to trigger the construction `\exp:w`, which expands `\operand:w`, defined to read what follows and make a number out of it, then leave `\exp_end:`, the number, and a semicolon in the input stream. Once `\operand:w` is done expanding, we obtain essentially

```
\exp_after:wN \add:ww \int_value:w 12345 ;
\exp:w \exp_end: 333444 ;
```

where in fact `\exp_after:wN` has already been expanded, `\int_value:w` has already seen `12345`, and `\exp:w` is still looking for a number. It finds `\exp_end:`, hence expands to nothing. Now, `\int_value:w` sees the `;`, which cannot be part of a number. The expansion stops, and we are left with

```
\add:ww 12345 ; 333444 ;
```

which can safely perform the addition by grabbing two arguments delimited by `;`.

If we were to continue parsing the expression, then the following number should also be cleaned up before the next use of a binary operation such as `\add:ww`. Just like `\int_value:w 12345 \exp_after:wN ;` expanded what follows once, we need `\add:ww` to do the calculation, and in the process to expand the following once. This is also true in our real application: all the functions of the form `\__fp_..._o:ww` expand what follows once. This comes at the cost of leaving tokens in the input stack, and we need to be careful not to waste this memory. All of our discussion above is nice but simplistic, as operations should not simply be performed in the order they appear.

71.1.2 Precedence and infix operators

The various operators we will encounter have different precedences, which influence the order of calculations: $1 + 2 \times 3 = 1 + (2 \times 3)$ because $\times$ has a higher precedence than $+$. The true analog of our macro `\operand:w` must thus take care of that. When looking for an operand, it needs to perform calculations until reaching an operator which has lower precedence than the one which called `\operand:w`. This means that `\operand:w` must know what the previous binary operator is, or rather, its precedence: we thus rename it `\operand:Nw`. Let us describe as an example how we plan to do the calculation `41-2^3*4+5`. More precisely we describe how to perform the first operation in this expression. Here, we abuse notations: the first argument of `\operand:Nw` should be an integer constant (`\c__fp_prec_plus_int`, ...) equal to the precedence of the given operator, not directly the operator itself.

- Clean up `41` and find `-`. We call `\operand:Nw -` to find the second operand.
- Clean up `2` and find `^`.
- Compare the precedences of `-` and `^`. Since the latter is higher, we need to compute the exponentiation. For this, find the second operand with a nested call to `\operand:Nw ^`.
- Clean up `3` and find `*`.
- Compare the precedences of `^` and `*`. Since the former is higher, `\operand:Nw ^` has found the second operand of the exponentiation, which is computed: $2^3 = 8$.
- We now have `41-8*4+5`, and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 8?
- Compare the precedences of `-` and `*`. Since the latter is higher, we are not done with 8. Call `\operand:Nw *` to find the second operand of the multiplication.
- Clean up `4`, and find `+`.
- Compare the precedences of `*` and `+`. Since the former is higher, `\operand:Nw *` has found the second operand of the multiplication, which is computed: $8 * 4 = 32$.
- We now have `41-32+5`, and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 32?
- Compare the precedences of `-` and `+`. Since they are equal, `\operand:Nw -` has found the second operand for the subtraction, which is computed: $41 - 32 = 9$.
- We now have `9+5`.

The procedure above stops short of performing all computations, but adding a surrounding call to `\operand:Nw` with a very low precedence ensures that all computations are performed before `\operand:Nw` is done. Adding a trailing marker with the same very low precedence prevents the surrounding `\operand:Nw` from going beyond the marker.

The pattern above to find an operand for a given operator, is to find one number and the next operator, then compare precedences to know if the next computation should be done. If it should, then perform it after finding its second operand, and look at the next operator, then compare precedences to know if the next computation should be done. This continues until we find that the next computation should not be done. Then, we stop.

We are now ready to get a bit more technical and describe which of the l3fp-parse functions correspond to each step above.

First, `\__fp_parse_operand:Nw` is the `\operand:Nw` function above, with small modifications due to expansion issues discussed later. We denote by ⟨*precedence*⟩ the argument of `\__fp_parse_operand:Nw`, that is, the precedence of the binary operator whose operand we are trying to find. The basic action is to read numbers from the input stream. This is done by `\__fp_parse_one:Nw`. A first approximation of this function is that it reads one ⟨*number*⟩, performing no computation, and finds the following binary ⟨*operator*⟩. Then it expands to

```
⟨number⟩
  \__fp_parse_infix_⟨operator⟩:N ⟨precedence⟩
```

expanding the `infix` auxiliary before leaving the above in the input stream.

We now explain the `infix` auxiliaries. We need some flexibility in how we treat the case of equal precedences: most often, the first operation encountered should be performed, such as `1-2-3` being computed as `(1-2)-3`, but `2^3^4` should be evaluated as `2^(3^4)` instead. For this reason, and to support the equivalence between `**` and `^` more easily, each binary operator is converted to a control sequence `\__fp_parse_infix_⟨operator⟩:N` when it is encountered for the first time. Instead of passing both precedences to a test function to do the comparison steps above, we pass the ⟨*precedence*⟩ (of the earlier operator) to the `infix` auxiliary for the following ⟨*operator*⟩, to know whether to perform the computation of the ⟨*operator*⟩. If it should not be performed, the `infix` auxiliary expands to

```
@ \use_none:n \__fp_parse_infix_⟨operator⟩:N
```

and otherwise it calls `\__fp_parse_operand:Nw` with the precedence of the ⟨*operator*⟩ to find its second operand ⟨$number_2$⟩ and the next ⟨$operator_2$⟩, and expands to

```
@ \__fp_parse_apply_binary:NwNwN
    ⟨operator⟩ ⟨number₂⟩
@ \__fp_parse_infix_⟨operator₂⟩:N
```

The `infix` function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand ⟨*number*⟩ is before the `infix` function in the input stream. This is why we stop the expansion here and give control to another function to close the loop.

A definition of `\__fp_parse_operand:Nw` ⟨*precedence*⟩ with some of the expansion control removed is

```
\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN ⟨precedence⟩
\exp:w \exp_end_continue_f:w
  \__fp_parse_one:Nw ⟨precedence⟩
```

This expands `\__fp_parse_one:Nw` ⟨*precedence*⟩ completely, which finds a number, wraps the next ⟨*operator*⟩ into an `infix` function, feeds this function the ⟨*precedence*⟩, and expands it, yielding either

```
\__fp_parse_continue:NwN ⟨precedence⟩
⟨number⟩ @
\use_none:n \__fp_parse_infix_⟨operator⟩:N
```

or

```
\__fp_parse_continue:NwN ⟨precedence⟩
⟨number⟩ @
\__fp_parse_apply_binary:NwNwN
  ⟨operator⟩ ⟨number₂⟩
@ \__fp_parse_infix_⟨operator₂⟩:N
```

The definition of `\__fp_parse_continue:NwN` is then very simple:

```
\cs_new:Npn \__fp_parse_continue:NwN #1#2@#3 { #3 #1 #2 @ }
```

In the first case, #3 is \use_none:n, yielding

```
\use_none:n ⟨precedence⟩ ⟨number⟩ @
\__fp_parse_infix_⟨operator⟩:N
```

then ⟨*number*⟩ @ __fp_parse_infix_⟨*operator*⟩:N. In the second case, #3 is __fp_-parse_apply_binary:NwNwN, whose role is to compute ⟨*number*⟩ ⟨*operator*⟩ ⟨*number*$_2$⟩ and to prepare for the next comparison of precedences: first we get

```
\__fp_parse_apply_binary:NwNwN
  ⟨precedence⟩ ⟨number⟩ @
  ⟨operator⟩ ⟨number2⟩
@ \__fp_parse_infix_⟨operator2⟩:N
```

then

```
\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN ⟨precedence⟩
\exp:w \exp_end_continue_f:w
\__fp_⟨operator⟩_o:ww ⟨number⟩ ⟨number2⟩
\exp:w \exp_end_continue_f:w
\__fp_parse_infix_⟨operator2⟩:N ⟨precedence⟩
```

where __fp_⟨*operator*⟩_o:ww computes ⟨*number*⟩ ⟨*operator*⟩ ⟨*number*$_2$⟩ and expands after the result, thus triggers the comparison of the precedence of the ⟨*operator*$_2$⟩ and the ⟨*precedence*⟩, continuing the loop.

We have introduced the most important functions here, and the next few paragraphs we describe various subtleties.

71.1.3 Prefix operators, parentheses, and functions

Prefix operators (unary -, +, !) and parentheses are taken care of by the same mechanism, and functions (sin, exp, etc.) as well. Finding the argument of the unary -, for instance, is very similar to grabbing the second operand of a binary infix operator, with a subtle precedence explained below. Once that operand is found, the operator can be applied to it (for the unary -, this simply flips the sign). A left parenthesis is just a prefix operator with a very low precedence equal to that of the closing parenthesis (which is treated as an infix operator, since it normally appears just after numbers), so that all computations are performed until the closing parenthesis. The prefix operator associated to the left parenthesis does not alter its argument, but it removes the closing parenthesis (with some checks).

Prefix operators are the reason why we only summarily described the function __fp_parse_one:Nw earlier. This function is responsible for reading in the input stream the first possible ⟨*number*⟩ and the next infix ⟨*operator*⟩. If what follows __fp_-parse_one:Nw ⟨*precedence*⟩ is a prefix operator, then we must find the operand of this prefix operator through a nested call to __fp_parse_operand:Nw with the appropriate precedence, then apply the operator to the operand found to yield the result of __fp_-parse_one:Nw. So far, all is simple.

The unary operators +, -, ! complicate things a little bit: -3**2 should be $-(3^2) = -9$, and not $(-3)^2 = 9$. This would easily be done by giving - a lower precedence, equal to that of the infix + and -. Unfortunately, this fails in cases such as 3**-2*4, yielding $3^{-2\times 4}$ instead of the correct $3^{-2}\times 4$. A second attempt would be to call __fp_-parse_operand:Nw with the ⟨*precedence*⟩ of the previous operator, but 0>-2+3 is then parsed as `0>-(2+3)`: the addition is performed because it binds more tightly than the comparison which precedes `-`. The correct approach is for a unary `-` to perform operations whose precedence is greater than both that of the previous operation, and that of the unary `-` itself. The unary `-` is given a precedence higher than multiplication and division. This does not lead to any surprising result, since $-(x/y) = (-x)/y$ and similarly for multiplication, and it reduces the number of nested calls to `\__fp_parse_operand:Nw`.

Functions are implemented as prefix operators with very high precedence, so that their argument is the first number that can possibly be built.

Note that contrarily to the `infix` functions discussed earlier, the `prefix` functions do perform tests on the previous ⟨*precedence*⟩ to decide whether to find an argument or not, since we know that we need a number, and must never stop there.

71.1.4 Numbers and reading tokens one by one

So far, we have glossed over one important point: what is a "number"? A number is typically given in the form ⟨*significand*⟩`e`⟨*exponent*⟩, where the ⟨*significand*⟩ is any non-empty string composed of decimal digits and at most one decimal separator (a period), the exponent "`e`⟨*exponent*⟩" is optional and is composed of an exponent mark `e` followed by a possibly empty string of signs `+` or `-` and a non-empty string of decimal digits. The ⟨*significand*⟩ can also be an integer, dimension, skip, or muskip variable, in which case dimensions are converted from points (or mu units) to floating points, and the ⟨*exponent*⟩ can also be an integer variable. Numbers can also be given as floating point variables, or as named constants such as `nan`, `inf` or `pi`. We may add more types in the future.

When `\__fp_parse_one:Nw` is looking for a "number", here is what happens.

- If the next token is a control sequence with the meaning of `\scan_stop:`, it can be: `\s__fp`, in which case our job is done, as what follows is an internal floating point number, or `\s__fp_expr_mark`, in which case the expression has come to an early end, as we are still looking for a number here, or something else, in which case we consider the control sequence to be a bad variable resulting from `c`-expansion.

- If the next token is a control sequence with a different meaning, we assume that it is a register, unpack it with `\tex_the:D`, and use its value (in `pt` for dimensions and skips, `mu` for muskips) as the ⟨*significand*⟩ of a number: we look for an exponent.

- If the next token is a digit, we remove any leading zeros, then read a significand larger than 1 if the next character is a digit, read a significand smaller than 1 if the next character is a period, or we have found a significand equal to 0 otherwise, and look for an exponent.

- If the next token is a letter, we collect more letters until the first non-letter: the resulting word may denote a function such as `asin`, a constant such as `pi` or be unknown. In the first case, we call `\__fp_parse_operand:Nw` to find the argument of the function, then apply the function, before declaring that we are done. Otherwise, we are done, either with the value of the constant, or with the value `nan` for unknown words.

- If the next token is anything else, we check whether it is a known prefix operator, in which case `\__fp_parse_operand:Nw` finds its operand. If it is not known, then either a number is missing (if the token is a known infix operator) or the token is simply invalid in floating point expressions.

Once a number is found, `\__fp_parse_one:Nw` also finds an infix operator. This goes as follows.

- If the next token is a control sequence, it could be the special marker `\s__fp_expr_mark`, and otherwise it is a case of juxtaposing numbers, such as `2\c_zero_int`, with an implied multiplication.
- If the next token is a letter, it is also a case of juxtaposition, as letters cannot be proper infix operators.
- Otherwise (including in the case of digits), if the token is a known infix operator, the appropriate `\__fp_infix_`⟨*operator*⟩`:N` function is built, and if it does not exist, we complain. In particular, the juxtaposition `\c_zero_int 2` is disallowed.

In the above, we need to test whether a character token `#1` is a digit:

```
\if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
  is a digit
\else:
  not a digit
\fi:
```

To exclude `0`, replace `9` by `10`. The use of `\token_to_str:N` ensures that a digit with any catcode is detected. To test if a character token is a letter, we need to work with its character code, testing if `` `#1`` lies in $[65, 90]$ (uppercase letters) or $[97, 112]$ (lowercase letters)

```
\if_int_compare:w \__fp_int_eval:w
    ( `#1 \if_int_compare:w `#1 > `Z - 32 \fi: ) / 26 = 3 \exp_stop_f:
  is a letter
\else:
  not a letter
\fi:
```

At all steps, we try to accept all category codes: when `#1` is kept to be used later, it is almost always converted to category code other through `\token_to_str:N`. More precisely, catcodes $\{3, 6, 7, 8, 11, 12\}$ should work without trouble, but not $\{1, 2, 4, 10, 13\}$, and of course $\{0, 5, 9\}$ cannot become tokens.

Floating point expressions should behave as much as possible like ε-TeX-based integer expressions and dimension expressions. In particular, `f`-expansion should be performed as the expression is read, token by token, forcing the expansion of protected macros, and ignoring spaces. One advantage of expanding at every step is that restricted expandable functions can then be used in floating point expressions just as they can be in other kinds of expressions. Problematically, spaces stop `f`-expansion: for instance, the macro `\X` below would not be expanded if we simply performed `f`-expansion.

```
\DeclareDocumentCommand {\test} {m} { \fp_eval:n {#1} }
\ExplSyntaxOff
\test { 1 + \X }
```

Of course, spaces typically do not appear in a code setting, but may very easily come in document-level input, from which some expressions may come. To avoid this problem, at every step, we do essentially what `\use:f` would do: take an argument, put it back in the input stream, then f-expand it. This is not a complete solution, since a macro's expansion could contain leading spaces which would stop the f-expansion before further macro calls are performed. However, in practice it should be enough: in particular, floating point numbers are correctly expanded to the underlying `\s__fp ...` structure. The f-expansion is performed by `\__fp_parse_expand:w`.

71.2 Main auxiliary functions

`\__fp_parse_operand:Nw`

`\exp:w \__fp_parse_operand:Nw ⟨precedence⟩ \__fp_parse_expand:w`

Reads the "...", performing every computation with a precedence higher than ⟨*precedence*⟩, then expands to

`⟨result⟩ @ \__fp_parse_infix_⟨operation⟩:N ...`

where the ⟨*operation*⟩ is the first operation with a lower precedence, possibly `end`, and the "..." start just after the ⟨*operation*⟩.

(*End of definition for* `\__fp_parse_operand:Nw`.)

`\__fp_parse_infix_+:N`

`\__fp_parse_infix_+:N ⟨precedence⟩ ...`

If `+` has a precedence higher than the ⟨*precedence*⟩, cleans up a second ⟨*operand*⟩ and finds the ⟨$operation_2$⟩ which follows, and expands to

`@ \__fp_parse_apply_binary:NwNwN + ⟨operand⟩ @ \__fp_parse_infix_⟨operation₂⟩:N ...`

Otherwise expands to

`@ \use_none:n \__fp_parse_infix_+:N ...`

A similar function exists for each infix operator.

(*End of definition for* `\__fp_parse_infix_+:N`.)

`\__fp_parse_one:Nw`

`\__fp_parse_one:Nw ⟨precedence⟩ ...`

Cleans up one or two operands depending on how the precedence of the next operation compares to the ⟨*precedence*⟩. If the following ⟨*operation*⟩ has a precedence higher than ⟨*precedence*⟩, expands to

`⟨operand1⟩ @ \__fp_parse_apply_binary:NwNwN ⟨operation⟩ ⟨operand2⟩ @`
`\__fp_parse_infix_⟨operation2⟩:N ...`

and otherwise expands to

`⟨operand⟩ @ \use_none:n \__fp_parse_infix_⟨operation⟩:N ...`

(*End of definition for* `\__fp_parse_one:Nw`.)

71.3 Helpers

__fp_parse_expand:w

`\exp:w \__fp_parse_expand:w` ⟨*tokens*⟩

This function must always come within a `\exp:w` expansion. The ⟨*tokens*⟩ should be the part of the expression that we have not yet read. This requires in particular closing all conditionals properly before expanding.

```
\cs_new:Npn \__fp_parse_expand:w #1 { \exp_end_continue_f:w #1 }
```

(*End of definition for* `\__fp_parse_expand:w`.)

__fp_parse_return_semicolon:w

This very odd function swaps its position with the following `\fi:` and removes `\__fp_parse_expand:w` normally responsible for expansion. That turns out to be useful.

```
\cs_new:Npn \__fp_parse_return_semicolon:w
    #1 \fi: \__fp_parse_expand:w { \fi: ; #1 }
```

(*End of definition for* `\__fp_parse_return_semicolon:w`.)

__fp_parse_digits_vii:N
__fp_parse_digits_vi:N
__fp_parse_digits_v:N
__fp_parse_digits_iv:N
__fp_parse_digits_iii:N
__fp_parse_digits_ii:N
__fp_parse_digits_i:N
__fp_parse_digits_:N

These functions must be called within an `\int_value:w` or `\__fp_int_eval:w` construction. The first token which follows must be `f`-expanded prior to calling those functions. The functions read tokens one by one, and output digits into the input stream, until meeting a non-digit, or up to a number of digits equal to their index. The full expansion is

⟨*digits*⟩ ; ⟨*filling 0*⟩ ; ⟨*length*⟩

where ⟨*filling 0*⟩ is a string of zeros such that ⟨*digits*⟩ ⟨*filling 0*⟩ has the length given by the index of the function, and ⟨*length*⟩ is the number of zeros in the ⟨*filling 0*⟩ string. Each function puts a digit into the input stream and calls the next function, until we find a non-digit. We are careful to pass the tested tokens through `\token_to_str:N` to normalize their category code.

```
\cs_set_protected:Npn \__fp_tmp:w #1 #2 #3
  {
    \cs_new:cpn { __fp_parse_digits_ #1 :N } ##1
      {
        \if_int_compare:w 9 < 1 \token_to_str:N ##1 \exp_stop_f:
          \token_to_str:N ##1 \exp_after:wN #2 \exp:w
        \else:
          \__fp_parse_return_semicolon:w #3 ##1
        \fi:
        \__fp_parse_expand:w
      }
  }
\__fp_tmp:w {vii}  \__fp_parse_digits_vi:N   { 0000000 ; 7 }
\__fp_tmp:w {vi}   \__fp_parse_digits_v:N    { 000000 ; 6 }
\__fp_tmp:w {v}    \__fp_parse_digits_iv:N   { 00000 ; 5 }
\__fp_tmp:w {iv}   \__fp_parse_digits_iii:N  { 0000 ; 4 }
\__fp_tmp:w {iii}  \__fp_parse_digits_ii:N   { 000 ; 3 }
\__fp_tmp:w {ii}   \__fp_parse_digits_i:N    { 00 ; 2 }
\__fp_tmp:w {i}    \__fp_parse_digits_:N     { 0 ; 1 }
\cs_new:Npn \__fp_parse_digits_:N { ; ; 0 }
```

(*End of definition for* `\__fp_parse_digits_vii:N` *and others.*)

71.4 Parsing one number

__fp_parse_one:Nw

This function finds one number, and packs the symbol which follows in an `\__fp_parse_infix_...` csname. `#1` is the previous ⟨*precedence*⟩, and `#2` the first token of the operand. We distinguish four cases: `#2` is equal to `\scan_stop:` in meaning, `#2` is a different control sequence, `#2` is a digit, and `#2` is something else (this last case is split further later). Despite the earlier `f`-expansion, `#2` may still be expandable if it was protected by `\exp_not:N`, as may happen with the LaTeX 2ε command `\protect`. Using a well placed `\reverse_if:N`, this case is sent to `\__fp_parse_one_fp:NN` which deals with it robustly.

```
\cs_new:Npn \__fp_parse_one:Nw #1 #2
  {
    \if_catcode:w \scan_stop: \exp_not:N #2
      \exp_after:wN \if_meaning:w \exp_not:N #2 #2 \else:
        \exp_after:wN \reverse_if:N
      \fi:
      \if_meaning:w \scan_stop: #2
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_one_fp:NN
      \else:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_one_register:NN
      \fi:
    \else:
      \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_one_digit:NN
      \else:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_one_other:NN
      \fi:
    \fi:
    #1 #2
  }
```

(*End of definition for* `\__fp_parse_one:Nw`.)

__fp_parse_one_fp:NN
__fp_exp_after_expr_mark_f:nw
__fp_exp_after_?_f:nw

This function receives a ⟨*precedence*⟩ and a control sequence equal to `\scan_stop:` in meaning. There are three cases.

- `\s__fp` starts a floating point number, and we call `\__fp_exp_after_f:nw`, which `f`-expands after the floating point.
- `\s__fp_expr_mark` is a premature end, we call `\__fp_exp_after_expr_mark_f:nw`, which triggers an `fp-early-end` error.
- For a control sequence not containing `\s__fp`, we call `\__fp_exp_after_?_f:nw`, causing a `bad-variable` error.

This scheme is extensible: additional types can be added by starting the variables with a scan mark of the form `\s__fp_`⟨*type*⟩ and defining `\__fp_exp_after_`⟨*type*⟩`_f:nw`. In all cases, we make sure that the second argument of `\__fp_parse_infix:NN` is correctly expanded. A special case only enabled in LaTeX 2ε is that if `\protect` is encountered then the error message mentions the control sequence which follows it rather than `\protect` itself. The test for LaTeX 2_ε uses `\@unexpandable@protect` rather than `\protect` because `\protect` is often `\scan_stop:` hence "does not exist".

```
\cs_new:Npn \__fp_parse_one_fp:NN #1
  {
    \__fp_exp_after_any_f:nw
      {
        \exp_after:wN \__fp_parse_infix:NN
        \exp_after:wN #1 \exp:w \__fp_parse_expand:w
      }
  }
\cs_new:Npn \__fp_exp_after_expr_mark_f:nw #1
  {
    \int_case:nnF { \exp_after:wN \use_i:nnn \use_none:nnn #1 }
      {
        \c__fp_prec_comma_int { }
        \c__fp_prec_tuple_int { }
        \c__fp_prec_end_int
          {
            \exp_after:wN \c__fp_empty_tuple_fp
            \exp:w \exp_end_continue_f:w
          }
      }
      {
        \msg_expandable_error:nn { fp } { early-end }
        \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
      }
    #1
  }
\cs_new:cpn { __fp_exp_after_?_f:nw } #1#2
  {
    \msg_expandable_error:nnn { kernel } { bad-variable }
      {#2}
    \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
  }
\cs_set_protected:Npn \__fp_tmp:w #1
  {
    \cs_if_exist:NT #1
      {
        \cs_gset:cpn { __fp_exp_after_?_f:nw } ##1##2
          {
            \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w ##1
            \str_if_eq:nnTF {##2} { \protect }
              {
                \cs_if_eq:NNTF ##2 #1 { \use_i:nn } { \use:n }
                {
                  \msg_expandable_error:nnn { fp }
                    { robust-cmd }
                }
              }
              {
                \msg_expandable_error:nnn { kernel }
                  { bad-variable } {##2}
              }
```

```
        }
      }
  }
\exp_args:Nc \__fp_tmp:w { @unexpandable@protect }
```

(*End of definition for* `\__fp_parse_one_fp:NN`*,* `\__fp_exp_after_expr_mark_f:nw`*, and* `\__fp_exp_after_?_f:nw`*.*)

`\__fp_parse_one_register:NN`
`\__fp_parse_one_register_aux:Nw`
`\__fp_parse_one_register_auxii:wwwNw`
`\__fp_parse_one_register_int:www`
`\__fp_parse_one_register_mu:www`
`\__fp_parse_one_register_dim:ww`

This is called whenever `#2` is a control sequence other than `\scan_stop:` in meaning. We special-case `\wd`, `\ht`, `\dp` (see later) and otherwise assume that it is a register, but carefully unpack it with `\tex_the:D` within braces. First, we find the exponent following `#2`. Then we unpack `#2` with `\tex_the:D`, and the `auxii` auxiliary distinguishes integer registers from dimensions/skips from muskips, according to the presence of a period and/or of `pt`. For integers, simply convert ⟨*value*⟩e⟨*exponent*⟩ to a floating point number with `\__fp_parse:n` (this is somewhat wasteful). For other registers, the decimal rounding provided by TeX does not accurately represent the binary value that it manipulates, so we extract this binary value as a number of scaled points with `\int_value:w \dim_to_decimal_in_sp:n {` ⟨*decimal value*⟩ `pt }`, and use an auxiliary of `\dim_to_fp:n`, which performs the multiplication by 2^{-16}, correctly rounded.

```
\cs_new:Npn \__fp_parse_one_register:NN #1#2
  {
    \exp_after:wN \__fp_parse_infix_after_operand:NwN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
      \__fp_parse_one_register_special:N #2
      \exp_after:wN \__fp_parse_one_register_aux:Nw
      \exp_after:wN #2
      \int_value:w
        \exp_after:wN \__fp_parse_exponent:N
        \exp:w \__fp_parse_expand:w
  }
\cs_new:Npe \__fp_parse_one_register_aux:Nw #1
  {
    \exp_not:n
      {
        \exp_after:wN \use:nn
        \exp_after:wN \__fp_parse_one_register_auxii:wwwNw
      }
    \exp_not:N \exp_after:wN { \exp_not:N \tex_the:D #1 }
      ; \exp_not:N \__fp_parse_one_register_dim:ww
      \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_mu:www
      . \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_int:www
      \s__fp_stop
  }
\exp_args:Nno \use:nn
  { \cs_new:Npn \__fp_parse_one_register_auxii:wwwNw #1 . #2 }
    { \tl_to_str:n { pt } #3 ; #4#5 \s__fp_stop }
    { #4 #1.#2; }
\exp_args:Nno \use:nn
  { \cs_new:Npn \__fp_parse_one_register_mu:www #1 }
    { \tl_to_str:n { mu } ; #2 ; }
    { \__fp_parse_one_register_dim:ww #1 ; }
\cs_new:Npn \__fp_parse_one_register_int:www #1; #2.; #3;
  { \__fp_parse:n { #1 e #3 } }
```

```
\cs_new:Npn \__fp_parse_one_register_dim:ww #1; #2;
  {
    \exp_after:wN \__fp_from_dim_test:ww
    \int_value:w #2 \exp_after:wN ,
    \int_value:w \dim_to_decimal_in_sp:n { #1 pt } ;
  }
```

(*End of definition for `\__fp_parse_one_register:NN` and others.*)

`\__fp_parse_one_register_special:N`
`\__fp_parse_one_register_math:NNw`
`\__fp_parse_one_register_wd:w`
`\__fp_parse_one_register_wd:Nw`

The `\wd`, `\dp`, `\ht` primitives expect an integer argument. We abuse the exponent parser to find the integer argument: simply include the exponent marker `e`. Once that "exponent" is found, use `\tex_the:D` to find the box dimension and then copy what we did for dimensions.

```
\cs_new:Npn \__fp_parse_one_register_special:N #1
  {
    \if_meaning:w \box_wd:N #1 \__fp_parse_one_register_wd:w \fi:
    \if_meaning:w \box_ht:N #1 \__fp_parse_one_register_wd:w \fi:
    \if_meaning:w \box_dp:N #1 \__fp_parse_one_register_wd:w \fi:
    \if_meaning:w \infty #1
      \__fp_parse_one_register_math:NNw \infty #1
    \fi:
    \if_meaning:w \pi #1
      \__fp_parse_one_register_math:NNw \pi #1
    \fi:
  }
\cs_new:Npn \__fp_parse_one_register_math:NNw
    #1#2#3#4 \__fp_parse_expand:w
  {
    #3
    \str_if_eq:nnTF {#1} {#2}
      {
        \msg_expandable_error:nnn
          { fp } { infty-pi } {#1}
        \c_nan_fp
      }
      { #4 \__fp_parse_expand:w }
  }
\cs_new:Npn \__fp_parse_one_register_wd:w
    #1#2 \exp_after:wN #3#4 \__fp_parse_expand:w
  {
    #1
    \exp_after:wN \__fp_parse_one_register_wd:Nw
    #4 \__fp_parse_expand:w e
  }
\cs_new:Npn \__fp_parse_one_register_wd:Nw #1#2 ;
  {
    \exp_after:wN \__fp_from_dim_test:ww
    \exp_after:wN 0 \exp_after:wN ,
    \int_value:w \dim_to_decimal_in_sp:n { #1 #2 } ;
  }
```

(*End of definition for `\__fp_parse_one_register_special:N` and others.*)

`\__fp_parse_one_digit:NN` A digit marks the beginning of an explicit floating point number. Once the number is found, we catch the case of overflow and underflow with `\__fp_sanitize:wN`, then __fp_parse_infix_after_operand:NwN expands __fp_parse_infix:NN after the number we find, to wrap the following infix operator as required. Finding the number itself begins by removing leading zeros: further steps are described later.

```
\cs_new:Npn \__fp_parse_one_digit:NN #1
  {
    \exp_after:wN \__fp_parse_infix_after_operand:NwN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
      \exp_after:wN \__fp_sanitize:wN
      \int_value:w \__fp_int_eval:w 0 \__fp_parse_trim_zeros:N
  }
```

(*End of definition for* __fp_parse_one_digit:NN.)

__fp_parse_one_other:NN For this function, #2 is a character token which is not a digit. If it is an ASCII letter, __fp_parse_letters:N beyond this one and give the result to __fp_parse_word:Nw. Otherwise, the character is assumed to be a prefix operator, and we build __fp_parse_prefix_⟨*operator*⟩:Nw.

```
\cs_new:Npn \__fp_parse_one_other:NN #1 #2
  {
    \if_int_compare:w
        \__fp_int_eval:w
          ( `#2 \if_int_compare:w `#2 > `Z - 32 \fi: ) / 26
        = 3 \exp_stop_f:
      \exp_after:wN \__fp_parse_word:Nw
      \exp_after:wN #1
      \exp_after:wN #2
      \exp:w \exp_after:wN \__fp_parse_letters:N
      \exp:w
    \else:
      \exp_after:wN \__fp_parse_prefix:NNN
      \exp_after:wN #1
      \exp_after:wN #2
      \cs:w
        __fp_parse_prefix_ \token_to_str:N #2 :Nw
        \exp_after:wN
      \cs_end:
      \exp:w
    \fi:
    \__fp_parse_expand:w
  }
```

(*End of definition for* __fp_parse_one_other:NN.)

__fp_parse_word:Nw
__fp_parse_letters:N
Finding letters is a simple recursion. Once __fp_parse_letters:N has done its job, we try to build a control sequence from the word #2. If it is a known word, then the corresponding action is taken, and otherwise, we complain about an unknown word, yield \c_nan_fp, and look for the following infix operator. Note that the unknown word could be a mistyped function as well as a mistyped constant, so there is no way to tell whether to look for arguments; we do not. The standard requires "inf" and "infinity" and "nan" to be recognized regardless of case, but we probably don't want to allow every l3fp word to have an arbitrary mixture of lower and upper case, so we test and use a differently-named control sequence.

```
\cs_new:Npn \__fp_parse_word:Nw #1#2;
  {
    \cs_if_exist_use:cF { __fp_parse_word_#2:N }
      {
        \cs_if_exist_use:cF
          { __fp_parse_caseless_ \str_casefold:n {#2} :N }
          {
            \msg_expandable_error:nnn
              { fp } { unknown-fp-word } {#2}
            \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
            \__fp_parse_infix:NN
          }
      }
      #1
  }
\cs_new:Npn \__fp_parse_letters:N #1
  {
    \exp_end_continue_f:w
    \if_int_compare:w
        \if_catcode:w \scan_stop: \exp_not:N #1
          0
        \else:
          \__fp_int_eval:w
            ( `#1 \if_int_compare:w `#1 > `Z - 32 \fi: ) / 26
        \fi:
        = 3 \exp_stop_f:
      \exp_after:wN #1
      \exp:w \exp_after:wN \__fp_parse_letters:N
      \exp:w
    \else:
      \__fp_parse_return_semicolon:w #1
    \fi:
    \__fp_parse_expand:w
  }
```

(*End of definition for* `\__fp_parse_word:Nw` *and* `\__fp_parse_letters:N`.)

`\__fp_parse_prefix:NNN`
`\__fp_parse_prefix_unknown:NNN`

For this function, `#1` is the previous ⟨*precedence*⟩, `#2` is the operator just seen, and `#3` is a control sequence which implements the operator if it is a known operator. If this control sequence is `\scan_stop:`, then the operator is in fact unknown. Either the expression is missing a number there (if the operator is valid as an infix operator), and we put `nan`, wrapping the infix operator in a csname as appropriate, or the character is simply invalid in floating point expressions, and we continue looking for a number, starting again from `\__fp_parse_one:Nw`.

```
\cs_new:Npn \__fp_parse_prefix:NNN #1#2#3
  {
    \if_meaning:w \scan_stop: #3
      \exp_after:wN \__fp_parse_prefix_unknown:NNN
      \exp_after:wN #2
    \fi:
    #3 #1
  }
\cs_new:Npn \__fp_parse_prefix_unknown:NNN #1#2#3
  {
```

```
    \cs_if_exist:cTF { __fp_parse_infix_ \token_to_str:N #1 :N }
      {
        \msg_expandable_error:nnn
          { fp } { missing-number } {#1}
        \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
        \__fp_parse_infix:NN #3 #1
      }
      {
        \msg_expandable_error:nnn
          { fp } { unknown-symbol } {#1}
        \__fp_parse_one:Nw #3
      }
  }
```

(*End of definition for* `\__fp_parse_prefix:NNN` *and* `\__fp_parse_prefix_unknown:NNN`.)

71.4.1 Numbers: trimming leading zeros

Numbers are parsed as follows: first we trim leading zeros, then if the next character is a digit, start reading a significand ≥ 1 with the set of functions `\__fp_parse_large`...; if it is a period, the significand is < 1; and otherwise it is zero. In the second case, trim additional zeros after the period, counting them for an exponent shift $\langle exp_1 \rangle < 0$, then read the significand with the set of functions `\__fp_parse_small`... Once the significand is read, read the exponent if `e` is present.

`\__fp_parse_trim_zeros:N`
`\__fp_parse_trim_end:w`

This function expects an already expanded token. It removes any leading zero, then distinguishes three cases: if the first non-zero token is a digit, then call `\__fp_parse_large:N` (the significand is ≥ 1); if it is `.`, then continue trimming zeros with `\__fp_parse_strim_zeros:N`; otherwise, our number is exactly zero, and we call `\__fp_parse_zero:` to take care of that case.

```
\cs_new:Npn \__fp_parse_trim_zeros:N #1
  {
    \if:w 0 \exp_not:N #1
      \exp_after:wN \__fp_parse_trim_zeros:N
      \exp:w
    \else:
      \if:w . \exp_not:N #1
        \exp_after:wN \__fp_parse_strim_zeros:N
        \exp:w
      \else:
        \__fp_parse_trim_end:w #1
      \fi:
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_trim_end:w #1 \fi: \fi: \__fp_parse_expand:w
  {
      \fi:
    \fi:
    \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
      \exp_after:wN \__fp_parse_large:N
    \else:
      \exp_after:wN \__fp_parse_zero:
```

```
    \fi:
    #1
  }
```

(*End of definition for* `\__fp_parse_trim_zeros:N` *and* `\__fp_parse_trim_end:w`.)

`\__fp_parse_strim_zeros:N`
`\__fp_parse_strim_end:w`

If we have removed all digits until a period (or if the body started with a period), then enter the "`small_trim`" loop which outputs -1 for each removed 0. Those -1 are added to an integer expression waiting for the exponent. If the first non-zero token is a digit, call `\__fp_parse_small:N` (our significand is smaller than 1), and otherwise, the number is an exact zero. The name `strim` stands for "small trim".

```
\cs_new:Npn \__fp_parse_strim_zeros:N #1
  {
    \if:w 0 \exp_not:N #1
      - 1
      \exp_after:wN \__fp_parse_strim_zeros:N \exp:w
    \else:
      \__fp_parse_strim_end:w #1
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_strim_end:w #1 \fi: \__fp_parse_expand:w
  {
    \fi:
    \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
      \exp_after:wN \__fp_parse_small:N
    \else:
      \exp_after:wN \__fp_parse_zero:
    \fi:
    #1
  }
```

(*End of definition for* `\__fp_parse_strim_zeros:N` *and* `\__fp_parse_strim_end:w`.)

`\__fp_parse_zero:`

After reading a significand of 0, find any exponent, then put a sign of `1` for `\__fp_sanitize:wN`, which removes everything and leaves an exact zero.

```
\cs_new:Npn \__fp_parse_zero:
  {
    \exp_after:wN ; \exp_after:wN 1
    \int_value:w \__fp_parse_exponent:N
  }
```

(*End of definition for* `\__fp_parse_zero:`.)

71.4.2 Number: small significand

`\__fp_parse_small:N`

This function is called after we have passed the decimal separator and removed all leading zeros from the significand. It is followed by a non-zero digit (with any catcode). The goal is to read up to 16 digits. But we can't do that all at once, because `\int_value:w` (which allows us to collect digits and continue expanding) can only go up to 9 digits. Hence we grab digits in two steps of 8 digits. Since `#1` is a digit, read seven more digits using `\__fp_parse_digits_vii:N`. The `small_leading` auxiliary leaves those digits in the `\int_value:w`, and grabs some more, or stops if there are no more digits. Then the pack_leading auxiliary puts the various parts in the appropriate order for the processing further up.

```
\cs_new:Npn \__fp_parse_small:N #1
  {
    \exp_after:wN \__fp_parse_pack_leading:NNNNNww
    \int_value:w \__fp_int_eval:w 1 \token_to_str:N #1
      \exp_after:wN \__fp_parse_small_leading:wwNN
      \int_value:w 1
        \exp_after:wN \__fp_parse_digits_vii:N
        \exp:w \__fp_parse_expand:w
  }
```

(*End of definition for* `\__fp_parse_small:N`.)

__fp_parse_small_leading:wwNN

`\__fp_parse_small_leading:wwNN 1 ⟨digits⟩ ; ⟨zeros⟩ ; ⟨number of zeros⟩`

We leave ⟨*digits*⟩ ⟨*zeros*⟩ in the input stream: the functions used to grab digits are such that this constitutes digits 1 through 8 of the significand. Then prepare to pack 8 more digits, with an exponent shift of zero (this shift is used in the case of a large significand). If #4 is a digit, leave it behind for the packing function, and read 6 more digits to reach a total of 15 digits: further digits are involved in the rounding. Otherwise put 8 zeros in to complete the significand, then look for an exponent.

```
\cs_new:Npn \__fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
  {
    #1 #2
    \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
    \exp_after:wN 0
    \int_value:w \__fp_int_eval:w 1
      \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
        \token_to_str:N #4
        \exp_after:wN \__fp_parse_small_trailing:wwNN
        \int_value:w 1
          \exp_after:wN \__fp_parse_digits_vi:N
          \exp:w
      \else:
        0000 0000 \__fp_parse_exponent:Nw #4
      \fi:
      \__fp_parse_expand:w
  }
```

(*End of definition for* `\__fp_parse_small_leading:wwNN`.)

__fp_parse_small_trailing:wwNN

`\__fp_parse_small_trailing:wwNN 1 ⟨digits⟩ ; ⟨zeros⟩ ; ⟨number of zeros⟩ ⟨next token⟩`

Leave digits 10 to 15 (arguments #1 and #2) in the input stream. If the ⟨*next token*⟩ is a digit, it is the 16th digit, we keep it, then the small_round auxiliary considers this digit and all further digits to perform the rounding: the function expands to nothing, to +0 or to +1. Otherwise, there is no 16-th digit, so we put a 0, and look for an exponent.

```
\cs_new:Npn \__fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
  {
    #1 #2
    \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
      \token_to_str:N #4
      \exp_after:wN \__fp_parse_small_round:NN
```

```
      \exp_after:wN #4
      \exp:w
    \else:
      0 \__fp_parse_exponent:Nw #4
    \fi:
    \__fp_parse_expand:w
  }
```

(*End of definition for* `\__fp_parse_small_trailing:wwNN`.)

`\__fp_parse_pack_trailing:NNNNNNww` `\__fp_parse_pack_leading:NNNNNww` `\__fp_parse_pack_carry:w`

Those functions are expanded after all the digits are found, we took care of the rounding, as well as the exponent. The last argument is the exponent. The previous five arguments are 8 digits which we pack in groups of 4, and the argument before that is 1, except in the rare case where rounding lead to a carry, in which case the argument is 2. The `trailing` function has an exponent shift as its first argument, which we add to the exponent found in the `e...` syntax. If the trailing digits cause a carry, the integer expression for the leading digits is incremented (`+1` in the code below). If the leading digits propagate this carry all the way up, the function `\__fp_parse_pack_carry:w` increments the exponent, and changes the significand from `0000...` to `1000...`: this is simple because such a carry can only occur to give rise to a power of 10.

```
\cs_new:Npn \__fp_parse_pack_trailing:NNNNNNww #1 #2 #3#4#5#6 #7; #8 ;
  {
    \if_meaning:w 2 #2 + 1 \fi:
    ; #8 + #1 ; {#3#4#5#6} {#7};
  }
\cs_new:Npn \__fp_parse_pack_leading:NNNNNww #1 #2#3#4#5 #6; #7;
  {
    + #7
    \if_meaning:w 2 #1 \__fp_parse_pack_carry:w \fi:
    ; 0 {#2#3#4#5} {#6}
  }
\cs_new:Npn \__fp_parse_pack_carry:w \fi: ; 0 #1
  { \fi: + 1 ; 0 {1000} }
```

(*End of definition for* `\__fp_parse_pack_trailing:NNNNNNww`, `\__fp_parse_pack_leading:NNNNNww`, *and* `\__fp_parse_pack_carry:w`.)

71.4.3 Number: large significand

Parsing a significand larger than 1 is a little bit more difficult than parsing small significands. We need to count the number of digits before the decimal separator, and add that to the final exponent. We also need to test for the presence of a dot each time we run out of digits, and branch to the appropriate `parse_small` function in those cases.

`\__fp_parse_large:N`

This function is followed by the first non-zero digit of a "large" significand (≥ 1). It is called within an integer expression for the exponent. Grab up to 7 more digits, for a total of 8 digits.

```
\cs_new:Npn \__fp_parse_large:N #1
  {
    \exp_after:wN \__fp_parse_large_leading:wwNN
    \int_value:w 1 \token_to_str:N #1
      \exp_after:wN \__fp_parse_digits_vii:N
      \exp:w \__fp_parse_expand:w
  }
```

(*End of definition for* `\__fp_parse_large:N`.)

`\__fp_parse_large_leading:wwNN`

`\__fp_parse_large_leading:wwNN 1` ⟨*digits*⟩ `;` ⟨*zeros*⟩ `;` ⟨*number of zeros*⟩ ⟨*next token*⟩

We shift the exponent by the number of digits in `#1`, namely the target number, 8, minus the ⟨*number of zeros*⟩ (number of digits missing). Then prepare to pack the 8 first digits. If the ⟨*next token*⟩ is a digit, read up to 6 more digits (digits 10 to 15). If it is a period, try to grab the end of our 8 first digits, branching to the `small` functions since the number of digit does not affect the exponent anymore. Finally, if this is the end of the significand, insert the ⟨*zeros*⟩ to complete the 8 first digits, insert 8 more, and look for an exponent.

```
\cs_new:Npn \__fp_parse_large_leading:wwNN 1 #1 ; #2; #3 #4
  {
    + \c__fp_half_prec_int - #3
    \exp_after:wN \__fp_parse_pack_leading:NNNNNww
    \int_value:w \__fp_int_eval:w 1 #1
      \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
        \exp_after:wN \__fp_parse_large_trailing:wwNN
        \int_value:w 1 \token_to_str:N #4
          \exp_after:wN \__fp_parse_digits_vi:N
          \exp:w
      \else:
        \if:w . \exp_not:N #4
          \exp_after:wN \__fp_parse_small_leading:wwNN
          \int_value:w 1
            \cs:w
              __fp_parse_digits_
              \__fp_int_to_roman:w #3
              :N \exp_after:wN
            \cs_end:
            \exp:w
        \else:
          #2
          \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
          \exp_after:wN 0
          \int_value:w 1 0000 0000
          \__fp_parse_exponent:Nw #4
        \fi:
      \fi:
      \__fp_parse_expand:w
  }
```

(*End of definition for* `\__fp_parse_large_leading:wwNN`.)

`\__fp_parse_large_trailing:wwNN`

`\__fp_parse_large_trailing:wwNN 1` ⟨*digits*⟩ `;` ⟨*zeros*⟩ `;` ⟨*number of zeros*⟩ ⟨*next token*⟩

We have just read 15 digits. If the ⟨*next token*⟩ is a digit, then the exponent shift caused by this block of 8 digits is 8, first argument to the `pack_trailing` function. We keep the ⟨*digits*⟩ and this 16-th digit, and find how this should be rounded using `\__fp_parse_large_round:NN`. Otherwise, the exponent shift is the number of ⟨*digits*⟩, 7 minus the ⟨*number of zeros*⟩, and we test for a decimal point. This case happens in `123451234512345.67` with exactly 15 digits before the decimal separator.

Then branch to the appropriate `small` auxiliary, grabbing a few more digits to complement the digits we already grabbed. Finally, if this is truly the end of the significand, look for an exponent after using the ⟨*zeros*⟩ and providing a 16-th digit of 0.

```
\cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
  {
    \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
      \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
      \exp_after:wN \c__fp_half_prec_int
      \int_value:w \__fp_int_eval:w 1 #1 \token_to_str:N #4
        \exp_after:wN \__fp_parse_large_round:NN
        \exp_after:wN #4
        \exp:w
    \else:
      \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
      \int_value:w \__fp_int_eval:w 7 - #3 \exp_stop_f:
      \int_value:w \__fp_int_eval:w 1 #1
        \if:w . \exp_not:N #4
          \exp_after:wN \__fp_parse_small_trailing:wwNN
          \int_value:w 1
            \cs:w
              __fp_parse_digits_
              \__fp_int_to_roman:w #3
              :N \exp_after:wN
            \cs_end:
            \exp:w
        \else:
          #2 0 \__fp_parse_exponent:Nw #4
        \fi:
    \fi:
    \__fp_parse_expand:w
  }
```

(*End of definition for* `\__fp_parse_large_trailing:wwNN`.)

71.4.4 Number: beyond 16 digits, rounding

`\__fp_parse_round_loop:N`
`\__fp_parse_round_up:N`

This loop is called when rounding a number (whether the mantissa is small or large). It should appear in an integer expression. This function reads digits one by one, until reaching a non-digit, and adds 1 to the integer expression for each digit. If all digits found are 0, the function ends the expression by `;0`, otherwise by `;1`. This is done by switching the loop to `round_up` at the first non-zero digit, thus we avoid to test whether digits are 0 or not once we see a first non-zero digit.

```
\cs_new:Npn \__fp_parse_round_loop:N #1
  {
    \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
      + 1
      \if:w 0 \token_to_str:N #1
        \exp_after:wN \__fp_parse_round_loop:N
        \exp:w
      \else:
        \exp_after:wN \__fp_parse_round_up:N
        \exp:w
      \fi:
```

```
    \else:
      \__fp_parse_return_semicolon:w 0 #1
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_round_up:N #1
  {
    \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
      + 1
      \exp_after:wN \__fp_parse_round_up:N
      \exp:w
    \else:
      \__fp_parse_return_semicolon:w 1 #1
    \fi:
    \__fp_parse_expand:w
  }
```

(*End of definition for* `\__fp_parse_round_loop:N` *and* `\__fp_parse_round_up:N`.)

`\__fp_parse_round_after:wN` After the loop `\__fp_parse_round_loop:N`, this function fetches an exponent with `\__fp_parse_exponent:N`, and combines it with the number of digits counted by `\__fp_parse_round_loop:N`. At the same time, the result `0` or `1` is added to the surrounding integer expression.

```
\cs_new:Npn \__fp_parse_round_after:wN #1; #2
  {
    + #2 \exp_after:wN ;
    \int_value:w \__fp_int_eval:w #1 + \__fp_parse_exponent:N
  }
```

(*End of definition for* `\__fp_parse_round_after:wN`.)

`\__fp_parse_small_round:NN`
`\__fp_parse_round_after:wN`

Here, `#1` is the digit that we are currently rounding (we only care whether it is even or odd). If `#2` is not a digit, then fetch an exponent and expand to `;`⟨*exponent*⟩ only. Otherwise, we expand to `+0` or `+1`, then `;`⟨*exponent*⟩. To decide which, call `\__fp_round_s:NNNw` to know whether to round up, giving it as arguments a sign 0 (all explicit numbers are positive), the digit `#1` to round, the first following digit `#2`, and either `+0` or `+1` depending on whether the following digits are all zero or not. This last argument is obtained by `\__fp_parse_round_loop:N`, whose number of digits we discard by multiplying it by 0. The exponent which follows the number is also fetched by `\__fp_parse_round_after:wN`.

```
\cs_new:Npn \__fp_parse_small_round:NN #1#2
  {
    \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
      +
      \exp_after:wN \__fp_round_s:NNNw
      \exp_after:wN 0
      \exp_after:wN #1
      \exp_after:wN #2
      \int_value:w \__fp_int_eval:w
        \exp_after:wN \__fp_parse_round_after:wN
        \int_value:w \__fp_int_eval:w 0 * \__fp_int_eval:w 0
          \exp_after:wN \__fp_parse_round_loop:N
          \exp:w
    \else:
```

```
      \__fp_parse_exponent:Nw #2
    \fi:
    \__fp_parse_expand:w
  }
```

(*End of definition for* `\__fp_parse_small_round:NN` *and* `\__fp_parse_round_after:wN`.)

`\__fp_parse_large_round:NN`
`\__fp_parse_large_round_test:NN`
`\__fp_parse_large_round_aux:wNN`

Large numbers are harder to round, as there may be a period in the way. Again, `#1` is the digit that we are currently rounding (we only care whether it is even or odd). If there are no more digits (`#2` is not a digit), then we must test for a period: if there is one, then switch to the rounding function for small significands, otherwise fetch an exponent. If there are more digits (`#2` is a digit), then round, checking with `\__fp_parse_round_loop:N` if all further digits vanish, or some are non-zero. This loop is not enough, as it is stopped by a period. After the loop, the `aux` function tests for a period: if it is present, then we must continue looking for digits, this time discarding the number of digits we find.

```
\cs_new:Npn \__fp_parse_large_round:NN #1#2
  {
    \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
      +
      \exp_after:wN \__fp_round_s:NNNw
      \exp_after:wN 0
      \exp_after:wN #1
      \exp_after:wN #2
      \int_value:w \__fp_int_eval:w
        \exp_after:wN \__fp_parse_large_round_aux:wNN
        \int_value:w \__fp_int_eval:w 1
          \exp_after:wN \__fp_parse_round_loop:N
    \else: %^^A could be dot, or e, or other
      \exp_after:wN \__fp_parse_large_round_test:NN
      \exp_after:wN #1
      \exp_after:wN #2
    \fi:
  }
\cs_new:Npn \__fp_parse_large_round_test:NN #1#2
  {
    \if:w . \exp_not:N #2
      \exp_after:wN \__fp_parse_small_round:NN
      \exp_after:wN #1
      \exp:w
    \else:
      \__fp_parse_exponent:Nw #2
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_large_round_aux:wNN #1 ; #2 #3
  {
    + #2
    \exp_after:wN \__fp_parse_round_after:wN
    \int_value:w \__fp_int_eval:w #1
      \if:w . \exp_not:N #3
        + 0 * \__fp_int_eval:w 0
          \exp_after:wN \__fp_parse_round_loop:N
          \exp:w \exp_after:wN \__fp_parse_expand:w
```

```
      \else:
        \exp_after:wN ;
        \exp_after:wN 0
        \exp_after:wN #3
      \fi:
  }
```

(*End of definition for* `\__fp_parse_large_round:NN`*,* `\__fp_parse_large_round_test:NN`*, and* `\__fp_parse_large_round_aux:wNN`*.*)

71.4.5 Number: finding the exponent

Expansion is a little bit tricky here, in part because we accept input where multiplication is implicit.

```
\__fp_parse:n { 3.2 erf(0.1) }
\__fp_parse:n { 3.2 e\l_my_int }
\__fp_parse:n { 3.2 \c_pi_fp }
```

The first case indicates that just looking one character ahead for an "`e`" is not enough, since we would mistake the function `erf` for an exponent of "`rf`". An alternative would be to look two tokens ahead and check if what follows is a sign or a digit, considering in that case that we must be finding an exponent. But taking care of the second case requires that we unpack registers after `e`. However, blindly expanding the two tokens ahead completely would break the third example (unpacking is even worse). Indeed, in the course of reading 3.2, `\c_pi_fp` is expanded to `\s__fp \__fp_chk:w 1 0 {-1} {3141}` $\cdots$ `;` and `\s__fp` stops the expansion. Expanding two tokens ahead would then force the expansion of `\__fp_chk:w` (despite it being protected), and that function tries to produce an error.

What can we do? Really, the reason why this last case breaks is that just as TeX does, we should read ahead as little as possible. Here, the only case where there may be an exponent is if the first token ahead is `e`. Then we expand (and possibly unpack) the second token.

`\__fp_parse_exponent:Nw`

This auxiliary is convenient to smuggle some material through `\fi:` ending conditional processing. We place those `\fi:` (argument `#2`) at a very odd place because this allows us to insert `\__fp_int_eval:w` ... there if needed.

```
\cs_new:Npn \__fp_parse_exponent:Nw #1 #2 \__fp_parse_expand:w
  {
    \exp_after:wN ;
    \int_value:w #2 \__fp_parse_exponent:N #1
  }
```

(*End of definition for* `\__fp_parse_exponent:Nw`*.*)

`\__fp_parse_exponent:N`
`\__fp_parse_exponent_aux:NN`

This function should be called within an `\int_value:w` expansion (or within an integer expression). It leaves digits of the exponent behind it in the input stream, and terminates the expansion with a semicolon. If there is no `e` (or `E`), leave an exponent of 0. If there is an `e` or `E`, expand the next token to run some tests on it. The first rough test is that if the character code of `#1` is greater than that of `9` (largest code valid for an exponent, less than any code valid for an identifier), there was in fact no exponent; otherwise, we search for the sign of the exponent.

```
\cs_new:Npn \__fp_parse_exponent:N #1
```

```
  {
    \if:w e \if:w E \exp_not:N #1 e \else: \exp_not:N #1 \fi:
      \exp_after:wN \__fp_parse_exponent_aux:NN
      \exp_after:wN #1
      \exp:w
    \else:
      0 \__fp_parse_return_semicolon:w #1
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_exponent_aux:NN #1#2
  {
    \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #2
                0 \else: `#2 \fi: > `9 \exp_stop_f:
      0 \exp_after:wN ; \exp_after:wN #1
    \else:
      \exp_after:wN \__fp_parse_exponent_sign:N
    \fi:
    #2
  }
```

(*End of definition for* `\__fp_parse_exponent:N` *and* `\__fp_parse_exponent_aux:NN`.)

`\__fp_parse_exponent_sign:N`

Read signs one by one (if there is any).

```
\cs_new:Npn \__fp_parse_exponent_sign:N #1
  {
    \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
      \exp_after:wN \__fp_parse_exponent_sign:N
      \exp:w \exp_after:wN \__fp_parse_expand:w
    \else:
      \exp_after:wN \__fp_parse_exponent_body:N
      \exp_after:wN #1
    \fi:
  }
```

(*End of definition for* `\__fp_parse_exponent_sign:N`.)

`\__fp_parse_exponent_body:N`

An exponent can be an explicit integer (most common case), or various other things (most of which are invalid).

```
\cs_new:Npn \__fp_parse_exponent_body:N #1
  {
    \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
      \token_to_str:N #1
      \exp_after:wN \__fp_parse_exponent_digits:N
      \exp:w
    \else:
      \__fp_parse_exponent_keep:NTF #1
        { \__fp_parse_return_semicolon:w #1 }
        {
          \exp_after:wN ;
          \exp:w
        }
    \fi:
    \__fp_parse_expand:w
  }
```

(*End of definition for* `\__fp_parse_exponent_body:N`.)

`\__fp_parse_exponent_digits:N`

Read digits one by one, and leave them behind in the input stream. When finding a non-digit, stop, and insert a semicolon. Note that we do not check for overflow of the exponent, hence there can be a TeX error. It is mostly harmless, except when parsing `0e9876543210`, which should be a valid representation of 0, but is not.

```
\cs_new:Npn \__fp_parse_exponent_digits:N #1
  {
    \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
      \token_to_str:N #1
      \exp_after:wN \__fp_parse_exponent_digits:N
      \exp:w
    \else:
      \__fp_parse_return_semicolon:w #1
    \fi:
    \__fp_parse_expand:w
  }
```

(*End of definition for* `\__fp_parse_exponent_digits:N`.)

`\__fp_parse_exponent_keep:NTF`

This is the last building block for parsing exponents. The argument `#1` is already fully expanded, and neither `+` nor `-` nor a digit. It can be:

- `\s__fp`, marking the start of an internal floating point, invalid here;
- another control sequence equal to `\relax`, probably a bad variable;
- a register: in this case we make sure that it is an integer register, not a dimension;
- a character other than `+`, `-` or digits, again, an error.

```
\prg_new_conditional:Npnn \__fp_parse_exponent_keep:N #1 { TF }
  {
    \if_catcode:w \scan_stop: \exp_not:N #1
      \if_meaning:w \scan_stop: #1
        \if:w 0 \__fp_str_if_eq:nn { \s__fp } { \exp_not:N #1 }
          0
          \msg_expandable_error:nnn
            { fp } { after-e } { floating~point~ }
          \prg_return_true:
        \else:
          0
          \msg_expandable_error:nnn
            { kernel } { bad-variable } {#1}
          \prg_return_false:
        \fi:
      \else:
        \if:w 0 \__fp_str_if_eq:nn { \int_value:w #1 } { \tex_the:D #1 }
          \int_value:w #1
        \else:
          0
          \msg_expandable_error:nnn
            { fp } { after-e } { dimension~#1 }
        \fi:
        \prg_return_false:
```

```
      \fi:
    \else:
      0
      \msg_expandable_error:nnn
        { fp } { missing } { exponent }
      \prg_return_true:
    \fi:
  }
```

(*End of definition for* `\__fp_parse_exponent_keep:NTF`.)

71.5 Constants, functions and prefix operators

71.5.1 Prefix operators

`\__fp_parse_prefix_+:Nw`

A unary `+` does nothing: we should continue looking for a number.

```
\cs_new_eq:cN { __fp_parse_prefix_+:Nw } \__fp_parse_one:Nw
```

(*End of definition for* `\__fp_parse_prefix_+:Nw`.)

`\__fp_parse_apply_function:NNNwN`

Here, `#1` is a precedence, `#2` is some extra data used by some functions, `#3` is *e.g.*, `\__fp_sin_o:w`, and expands once after the calculation, `#4` is the operand, and `#5` is a `\__fp_parse_infix_...:N` function. We feed the data `#2`, and the argument `#4`, to the function `#3`, which expands `\exp:w` thus the `infix` function `#5`.

```
\cs_new:Npn \__fp_parse_apply_function:NNNwN #1#2#3#4@#5
  {
    #3 #2 #4 @
    \exp:w \exp_end_continue_f:w #5 #1
  }
```

(*End of definition for* `\__fp_parse_apply_function:NNNwN`.)

`\__fp_parse_apply_unary:NNNwN`
`\__fp_parse_apply_unary_chk:NwNw`
`\__fp_parse_apply_unary_chk:nNNNw`
`\__fp_parse_apply_unary_type:NNN`
`\__fp_parse_apply_unary_error:NNw`

In contrast to `\__fp_parse_apply_function:NNNwN`, this checks that the operand `#4` is a single argument (namely there is a single `;`). We use the fact that any floating point starts with a "safe" token like `\s__fp`. If there is no argument produce the `fp-no-arg` error; if there are at least two produce `fp-multi-arg`. For the error message extract the mathematical function name (such as `sin`) from the expl3 function that computes it, such as `\__fp_sin_o:w`.

In addition, since there is a single argument we can dispatch on type and check that the resulting function exists. This catches things like `sin((1,2))` where it does not make sense to take the sine of a tuple.

```
\cs_new:Npn \__fp_parse_apply_unary:NNNwN #1#2#3#4@#5
  {
    \__fp_parse_apply_unary_chk:NwNw #4 @ ; . \s__fp_stop
    \__fp_parse_apply_unary_type:NNN
    #3 #2 #4 @
    \exp:w \exp_end_continue_f:w #5 #1
  }
\cs_new:Npn \__fp_parse_apply_unary_chk:NwNw #1#2 ; #3#4 \s__fp_stop
  {
    \if_meaning:w @ #3 \else:
      \token_if_eq_meaning:NNTF . #3
        { \__fp_parse_apply_unary_chk:nNNNNw { no } }
```

```
      { \__fp_parse_apply_unary_chk:nNNNNw { multi } }
    \fi:
  }
\cs_new:Npn \__fp_parse_apply_unary_chk:nNNNNw #1#2#3#4#5#6 @
  {
    #2
    \__fp_error:nffn { #1-arg } { \__fp_func_to_name:N #4 } { } { }
    \exp_after:wN #4 \exp_after:wN #5 \c_nan_fp @
  }
\cs_new:Npn \__fp_parse_apply_unary_type:NNN #1#2#3
  {
    \__fp_change_func_type:NNN #3 #1 \__fp_parse_apply_unary_error:NNw
    #2 #3
  }
\cs_new:Npn \__fp_parse_apply_unary_error:NNw #1#2#3 @
  { \__fp_invalid_operation_o:fw { \__fp_func_to_name:N #1 } #3 }
```

(*End of definition for* `\__fp_parse_apply_unary:NNNwN` *and others.*)

`\__fp_parse_prefix_-:Nw`
`\__fp_parse_prefix_!:Nw`

The unary `-` and boolean not are harder: we parse the operand using a precedence equal to the maximum of the previous precedence `##1` and the precedence `\c__fp_prec_not_int` of the unary operator, then call the appropriate `\__fp_`⟨*operation*⟩`_o:w` function, where the ⟨*operation*⟩ is `set_sign` or `not`.

```
\cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
  {
    \cs_new:cpn { __fp_parse_prefix_ #1 :Nw } ##1
      {
        \exp_after:wN \__fp_parse_apply_unary:NNNwN
        \exp_after:wN ##1
        \exp_after:wN #4
        \exp_after:wN #3
        \exp:w
        \if_int_compare:w #2 < ##1
          \__fp_parse_operand:Nw ##1
        \else:
          \__fp_parse_operand:Nw #2
        \fi:
        \__fp_parse_expand:w
      }
  }
\__fp_tmp:w - \c__fp_prec_not_int \__fp_set_sign_o:w 2
\__fp_tmp:w ! \c__fp_prec_not_int \__fp_not_o:w ?
```

(*End of definition for* `\__fp_parse_prefix_-:Nw` *and* `\__fp_parse_prefix_!:Nw`*.*)

`\__fp_parse_prefix_.:Nw`

Numbers which start with a decimal separator (a period) end up here. Of course, we do not look for an operand, but for the rest of the number. This function is very similar to `\__fp_parse_one_digit:NN` but calls `\__fp_parse_strim_zeros:N` to trim zeros after the decimal point, rather than the `trim_zeros` function for zeros before the decimal point.

```
\cs_new:cpn { __fp_parse_prefix_.:Nw } #1
  {
    \exp_after:wN \__fp_parse_infix_after_operand:NwN
    \exp_after:wN #1
```

```
    \exp:w \exp_end_continue_f:w
      \exp_after:wN \__fp_sanitize:wN
      \int_value:w \__fp_int_eval:w 0 \__fp_parse_strim_zeros:N
  }
```

(*End of definition for* `\__fp_parse_prefix_.:Nw`.)

`\__fp_parse_prefix_(:Nw`
`\__fp_parse_lparen_after:NwN`

The left parenthesis is treated as a unary prefix operator because it appears in exactly the same settings. If the previous precedence is `\c__fp_prec_func_int` we are parsing arguments of a function and commas should not build tuples; otherwise commas should build tuples. We distinguish these cases by precedence: `\c__fp_prec_comma_int` for the case of arguments, `\c__fp_prec_tuple_int` for the case of tuples. Once the operand is found, the `lparen_after` auxiliary makes sure that there was a closing parenthesis (otherwise it complains), and leaves in the input stream an operand, fetching the following infix operator.

```
\cs_new:cpn { __fp_parse_prefix_(:Nw } #1
  {
    \exp_after:wN \__fp_parse_lparen_after:NwN
    \exp_after:wN #1
    \exp:w
    \if_int_compare:w #1 = \c__fp_prec_func_int
      \__fp_parse_operand:Nw \c__fp_prec_comma_int
    \else:
      \__fp_parse_operand:Nw \c__fp_prec_tuple_int
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npe \__fp_parse_lparen_after:NwN #1#2 @ #3
  {
    \exp_not:N \token_if_eq_meaning:NNTF #3
      \exp_not:c { __fp_parse_infix_):N }
      {
        \exp_not:N \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
        \exp_not:N \exp_after:wN
        \exp_not:N \__fp_parse_infix_after_paren:NN
        \exp_not:N \exp_after:wN #1
        \exp_not:N \exp:w
        \exp_not:N \__fp_parse_expand:w
      }
      {
        \exp_not:N \msg_expandable_error:nnn
          { fp } { missing } { ) }
        \exp_not:N \tl_if_empty:nT {#2} \exp_not:N \c__fp_empty_tuple_fp
        #2 @
        \exp_not:N \use_none:n #3
      }
  }
```

(*End of definition for* `\__fp_parse_prefix_(:Nw` *and* `\__fp_parse_lparen_after:NwN`.)

`\__fp_parse_prefix_):Nw`

The right parenthesis can appear as a prefix in two similar cases: in an empty tuple or tuple ending with a comma, or in an empty argument list or argument list ending with a comma, such as in `max(1,2,)` or in `rand()`.

```
\cs_new:cpn { __fp_parse_prefix_):Nw } #1
```

```
  {
    \if_int_compare:w #1 = \c__fp_prec_comma_int
    \else:
      \if_int_compare:w #1 = \c__fp_prec_tuple_int
        \exp_after:wN \c__fp_empty_tuple_fp \exp:w
      \else:
        \msg_expandable_error:nnn
          { fp } { missing-number } { ) }
        \exp_after:wN \c_nan_fp \exp:w
      \fi:
      \exp_end_continue_f:w
    \fi:
    \__fp_parse_infix_after_paren:NN #1 )
  }
```

(*End of definition for* `\__fp_parse_prefix_):Nw`*.*)

71.5.2 Constants

`\__fp_parse_word_inf:N`
`\__fp_parse_word_nan:N`
`\__fp_parse_word_pi:N`
`\__fp_parse_word_deg:N`
`\__fp_parse_word_true:N`
`\__fp_parse_word_false:N`

Some words correspond to constant floating points. The floating point constant is left as a result of `\__fp_parse_one:Nw` after expanding `\__fp_parse_infix:NN`.

```
\cs_set_protected:Npn \__fp_tmp:w #1 #2
  {
    \cs_new:cpn { __fp_parse_word_#1:N }
      { \exp_after:wN #2 \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN }
  }
\__fp_tmp:w { inf } \c_inf_fp
\__fp_tmp:w { nan } \c_nan_fp
\__fp_tmp:w { pi  } \c_pi_fp
\__fp_tmp:w { deg } \c_one_degree_fp
\__fp_tmp:w { true } \c_one_fp
\__fp_tmp:w { false } \c_zero_fp
```

(*End of definition for* `\__fp_parse_word_inf:N` *and others.*)

`\__fp_parse_caseless_inf:N`
`\__fp_parse_caseless_infinity:N`
`\__fp_parse_caseless_nan:N`

Copies of `\__fp_parse_word_...:N` commands, to allow arbitrary case as mandated by the standard.

```
\cs_new_eq:NN \__fp_parse_caseless_inf:N \__fp_parse_word_inf:N
\cs_new_eq:NN \__fp_parse_caseless_infinity:N \__fp_parse_word_inf:N
\cs_new_eq:NN \__fp_parse_caseless_nan:N \__fp_parse_word_nan:N
```

(*End of definition for* `\__fp_parse_caseless_inf:N`*,* `\__fp_parse_caseless_infinity:N`*, and* `\__fp_parse_caseless_nan:N`*.*)

`\__fp_parse_word_pt:N`
`\__fp_parse_word_in:N`
`\__fp_parse_word_pc:N`
`\__fp_parse_word_cm:N`
`\__fp_parse_word_mm:N`
`\__fp_parse_word_dd:N`
`\__fp_parse_word_cc:N`
`\__fp_parse_word_nd:N`
`\__fp_parse_word_nc:N`
`\__fp_parse_word_bp:N`
`\__fp_parse_word_sp:N`

Dimension units are also floating point constants but their value is not stored as a floating point constant. We give the values explicitly here.

```
\cs_set_protected:Npn \__fp_tmp:w #1 #2
  {
    \cs_new:cpn { __fp_parse_word_#1:N }
      {
        \__fp_exp_after_f:nw { \__fp_parse_infix:NN }
        \s__fp \__fp_chk:w 10 #2 ;
      }
  }
```

```
\__fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
\__fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
\__fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
\__fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
\__fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
\__fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
\__fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
\__fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
\__fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
\__fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
\__fp_tmp:w {sp} { {-4} {1525} {8789} {0625} {0000} }
```

(*End of definition for* `\__fp_parse_word_pt:N` *and others.*)

`\__fp_parse_word_em:N`
`\__fp_parse_word_ex:N`

The font-dependent units `em` and `ex` must be evaluated on the fly. We reuse an auxiliary of `\dim_to_fp:n`.

```
\tl_map_inline:nn { {em} {ex} }
  {
    \cs_new:cpn { __fp_parse_word_#1:N }
      {
        \exp_after:wN \__fp_from_dim_test:ww
        \exp_after:wN 0 \exp_after:wN ,
        \int_value:w \dim_to_decimal_in_sp:n { 1 #1 } \exp_after:wN ;
        \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN
      }
  }
```

(*End of definition for* `\__fp_parse_word_em:N` *and* `\__fp_parse_word_ex:N`.)

71.5.3 Functions

`\__fp_parse_unary_function:NNN`
`\__fp_parse_function:NNN`

```
\cs_new:Npn \__fp_parse_unary_function:NNN #1#2#3
  {
    \exp_after:wN \__fp_parse_apply_unary:NNNwN
    \exp_after:wN #3
    \exp_after:wN #2
    \exp_after:wN #1
    \exp:w
    \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_function:NNN #1#2#3
  {
    \exp_after:wN \__fp_parse_apply_function:NNNwN
    \exp_after:wN #3
    \exp_after:wN #2
    \exp_after:wN #1
    \exp:w
    \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
  }
```

(*End of definition for* `\__fp_parse_unary_function:NNN` *and* `\__fp_parse_function:NNN`.)

71.6 Main functions

__fp_parse:n
__fp_parse_o:n
__fp_parse_after:ww

Start an \exp:w expansion so that __fp_parse:n expands in two steps. The __fp_-parse_operand:Nw function performs computations until reaching an operation with precedence \c__fp_prec_end_int or less, namely, the end of the expression. The marker \s__fp_expr_mark indicates that the next token is an already parsed version of an infix operator, and __fp_parse_infix_end:N has infinitely negative precedence. Finally, clean up a (well-defined) set of extra tokens and stop the initial expansion with \exp_-end:.

```
\cs_new:Npn \__fp_parse:n #1
  {
    \exp:w
      \exp_after:wN \__fp_parse_after:ww
      \exp:w
        \__fp_parse_operand:Nw \c__fp_prec_end_int
        \__fp_parse_expand:w #1
        \s__fp_expr_mark \__fp_parse_infix_end:N
      \s__fp_expr_stop
    \exp_end:
  }
\cs_new:Npn \__fp_parse_after:ww
    #1@ \__fp_parse_infix_end:N \s__fp_expr_stop #2 { #2 #1 }
\cs_new:Npn \__fp_parse_o:n #1
  {
    \exp:w
      \exp_after:wN \__fp_parse_after:ww
      \exp:w
        \__fp_parse_operand:Nw \c__fp_prec_end_int
        \__fp_parse_expand:w #1
        \s__fp_expr_mark \__fp_parse_infix_end:N
      \s__fp_expr_stop
    {
      \exp_end_continue_f:w
      \__fp_exp_after_any_f:nw { \exp_after:wN \exp_stop_f: }
    }
  }
```

(*End of definition for* __fp_parse:n *,* __fp_parse_o:n *, and* __fp_parse_after:ww*.*)

__fp_parse_operand:Nw
__fp_parse_continue:NwN

This is just a shorthand which sets up both __fp_parse_continue:NwN and __fp_-parse_one:Nw with the same precedence. Note the trailing \exp:w.

```
\cs_new:Npn \__fp_parse_operand:Nw #1
  {
    \exp_end_continue_f:w
    \exp_after:wN \__fp_parse_continue:NwN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
    \exp_after:wN \__fp_parse_one:Nw
    \exp_after:wN #1
    \exp:w
  }
\cs_new:Npn \__fp_parse_continue:NwN #1 #2 @ #3 { #3 #1 #2 @ }
```

(*End of definition for* __fp_parse_operand:Nw *and* __fp_parse_continue:NwN*.*)

__fp_parse_apply_binary:NwNwN
__fp_parse_apply_binary_chk:NN
__fp_parse_apply_binary_error:NNN

Receives ⟨*precedence*⟩ ⟨*operand*$_1$⟩ @ ⟨*operation*⟩ ⟨*operand*$_2$⟩ @ ⟨*infix command*⟩. Builds the appropriate call to the ⟨*operation*⟩ #3, dispatching on both types. If the resulting control sequence does not exist, the operation is not allowed.

This is redefined in l3fp-extras.

```
\cs_new:Npn \__fp_parse_apply_binary:NwNwN #1 #2#3@ #4 #5#6@ #7
  {
    \exp_after:wN \__fp_parse_continue:NwN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
      \exp_after:wN \__fp_parse_apply_binary_chk:NN
        \cs:w
          __fp
          \__fp_type_from_scan:N #2
          _#4
          \__fp_type_from_scan:N #5
          _o:ww
        \cs_end:
        #4
      #2#3 #5#6
    \exp:w \exp_end_continue_f:w #7 #1
  }
\cs_new:Npn \__fp_parse_apply_binary_chk:NN #1#2
  {
    \if_meaning:w \scan_stop: #1
      \__fp_parse_apply_binary_error:NNN #2
    \fi:
    #1
  }
\cs_new:Npn \__fp_parse_apply_binary_error:NNN #1#2#3
  {
    #2
    \__fp_invalid_operation_o:Nww #1
  }
```

(*End of definition for* __fp_parse_apply_binary:NwNwN*,* __fp_parse_apply_binary_chk:NN*, and* __fp_parse_apply_binary_error:NNN*.*)

__fp_binary_type_o:Nww
__fp_binary_rev_type_o:Nww

Applies the operator #1 to its two arguments, dispatching according to their types, and expands once after the result. The rev version swaps its arguments before doing this.

```
\cs_new:Npn \__fp_binary_type_o:Nww #1 #2#3 ; #4
  {
    \exp_after:wN \__fp_parse_apply_binary_chk:NN
      \cs:w
        __fp
        \__fp_type_from_scan:N #2
        _ #1
        \__fp_type_from_scan:N #4
        _o:ww
      \cs_end:
      #1
    #2 #3 ; #4
  }
\cs_new:Npn \__fp_binary_rev_type_o:Nww #1 #2#3 ; #4#5 ;
  {
```

```
    \exp_after:wN \__fp_parse_apply_binary_chk:NN
      \cs:w
        __fp
        \__fp_type_from_scan:N #4
        _ #1
        \__fp_type_from_scan:N #2
        _o:ww
      \cs_end:
      #1
    #4 #5 ; #2 #3 ;
  }
```

(*End of definition for* `\__fp_binary_type_o:Nww` *and* `\__fp_binary_rev_type_o:Nww`.)

71.7 Infix operators

`\__fp_parse_infix_after_operand:NwN`

```
\cs_new:Npn \__fp_parse_infix_after_operand:NwN #1 #2;
  {
    \__fp_exp_after_f:nw { \__fp_parse_infix:NN #1 }
    #2;
  }
\cs_new:Npn \__fp_parse_infix:NN #1 #2
  {
    \if_catcode:w \scan_stop: \exp_not:N #2
      \if:w 0 \__fp_str_if_eq:nn { \s__fp_expr_mark } { \exp_not:N #2 }
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_infix_mark:NNN
      \else:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_infix_juxt:N
      \fi:
    \else:
      \if_int_compare:w
          \__fp_int_eval:w
            ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
          = 3 \exp_stop_f:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_infix_juxt:N
      \else:
        \exp_after:wN \__fp_parse_infix_check:NNN
        \cs:w
          __fp_parse_infix_ \token_to_str:N #2 :N
          \exp_after:wN \exp_after:wN \exp_after:wN
        \cs_end:
      \fi:
    \fi:
    #1
    #2
  }
\cs_new:Npn \__fp_parse_infix_check:NNN #1#2#3
  {
    \if_meaning:w \scan_stop: #1
```

```
      \msg_expandable_error:nnn
        { fp } { missing } { * }
      \exp_after:wN \__fp_parse_infix_mul:N
      \exp_after:wN #2
      \exp_after:wN #3
    \else:
      \exp_after:wN #1
      \exp_after:wN #2
      \exp:w \exp_after:wN \__fp_parse_expand:w
    \fi:
  }
```

(*End of definition for* `\__fp_parse_infix_after_operand:NwN`.)

`\__fp_parse_infix_after_paren:NN` Variant of `\__fp_parse_infix:NN` for use after a closing parenthesis. The only difference is that `\__fp_parse_infix_juxt:N` is replaced by `\__fp_parse_infix_mul:N`.

```
\cs_new:Npn \__fp_parse_infix_after_paren:NN #1 #2
  {
    \if_catcode:w \scan_stop: \exp_not:N #2
      \if:w 0 \__fp_str_if_eq:nn { \s__fp_expr_mark } { \exp_not:N #2 }
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_infix_mark:NNN
      \else:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_infix_mul:N
      \fi:
    \else:
      \if_int_compare:w
          \__fp_int_eval:w
            ( `#2 \if_int_compare:w `#2 > `Z - 32 \fi: ) / 26
          = 3 \exp_stop_f:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_infix_mul:N
      \else:
        \exp_after:wN \__fp_parse_infix_check:NNN
        \cs:w
          __fp_parse_infix_ \token_to_str:N #2 :N
          \exp_after:wN \exp_after:wN \exp_after:wN
        \cs_end:
      \fi:
    \fi:
    #1
    #2
  }
```

(*End of definition for* `\__fp_parse_infix_after_paren:NN`.)

71.7.1 Closing parentheses and commas

`\__fp_parse_infix_mark:NNN` As an infix operator, `\s__fp_expr_mark` means that the next token (`#3`) has already gone through `\__fp_parse_infix:NN` and should be provided the precedence `#1`. The scan mark `#2` is discarded.

```
\cs_new:Npn \__fp_parse_infix_mark:NNN #1#2#3 { #3 #1 }
```

(*End of definition for* `\__fp_parse_infix_mark:NNN`.)

`\__fp_parse_infix_end:N`

This one is a little bit odd: force every previous operator to end, regardless of the precedence.

```
\cs_new:Npn \__fp_parse_infix_end:N #1
  { @ \use_none:n \__fp_parse_infix_end:N }
```

(*End of definition for* `\__fp_parse_infix_end:N`.)

`\__fp_parse_infix_):N`

This is very similar to `\__fp_parse_infix_end:N`, complaining about an extra closing parenthesis if the previous operator was the beginning of the expression, with precedence `\c__fp_prec_end_int`.

```
\cs_set_protected:Npn \__fp_tmp:w #1
  {
    \cs_new:Npn #1 ##1
      {
        \if_int_compare:w ##1 > \c__fp_prec_end_int
          \exp_after:wN @
          \exp_after:wN \use_none:n
          \exp_after:wN #1
        \else:
          \msg_expandable_error:nnn { fp } { extra } { ) }
          \exp_after:wN \__fp_parse_infix:NN
          \exp_after:wN ##1
          \exp:w \exp_after:wN \__fp_parse_expand:w
        \fi:
      }
  }
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_):N }
```

(*End of definition for* `\__fp_parse_infix_):N`.)

`\__fp_parse_infix_,:N`
`\__fp_parse_infix_comma:w`
`\__fp_parse_apply_comma:NwNwN`

As for other infix operations, if the previous operations has higher precedence the comma waits. Otherwise we call `\__fp_parse_operand:Nw` to read more comma-delimited arguments that `\__fp_parse_infix_comma:w` simply concatenates into a `@`-delimited array. The first comma in a tuple that is not a function argument is distinguished: in that case call `\__fp_parse_apply_comma:NwNwN` whose job is to convert the first item of the tuple and an array of the remaining items into a tuple. In contrast to `\__fp_parse_apply_binary:NwNwN` this function's operands are not single-object arrays.

```
\cs_set_protected:Npn \__fp_tmp:w #1
  {
    \cs_new:Npn #1 ##1
      {
        \if_int_compare:w ##1 > \c__fp_prec_comma_int
          \exp_after:wN @
          \exp_after:wN \use_none:n
          \exp_after:wN #1
        \else:
          \if_int_compare:w ##1 < \c__fp_prec_comma_int
            \exp_after:wN @
            \exp_after:wN \__fp_parse_apply_comma:NwNwN
            \exp_after:wN ,
            \exp:w
          \else:
```

```
            \exp_after:wN \__fp_parse_infix_comma:w
            \exp:w
          \fi:
          \__fp_parse_operand:Nw \c__fp_prec_comma_int
          \exp_after:wN \__fp_parse_expand:w
        \fi:
      }
  }
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_,:N }
\cs_new:Npn \__fp_parse_infix_comma:w #1 @
  { #1 @ \use_none:n }
\cs_new:Npn \__fp_parse_apply_comma:NwNwN #1 #2@ #3 #4@ #5
  {
    \exp_after:wN \__fp_parse_continue:NwN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
    \__fp_exp_after_tuple_f:nw { }
      \s__fp_tuple \__fp_tuple_chk:w { #2 #4 } ;
    #5 #1
  }
```

(*End of definition for* `\__fp_parse_infix_,:N`*,* `\__fp_parse_infix_comma:w`*, and* `\__fp_parse_apply_-comma:NwNwN`*.*)

71.7.2 Usual infix operators

`\__fp_parse_infix_+:N`
`\__fp_parse_infix_-:N`
`\__fp_parse_infix_juxt:N`
`\__fp_parse_infix_/:N`
`\__fp_parse_infix_mul:N`
`\__fp_parse_infix_and:N`
`\__fp_parse_infix_or:N`
`\__fp_parse_infix_^:N`

As described in the "work plan", each infix operator has an associated `\..._infix_...` function, a computing function, and precedence, given as arguments to `\__fp_tmp:w`. Using the general mechanism for arithmetic operations. The power operation must be associative in the opposite order from all others. For this, we use two distinct precedences.

```
\cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
  {
    \cs_new:Npn #1 ##1
      {
        \if_int_compare:w ##1 < #3
          \exp_after:wN @
          \exp_after:wN \__fp_parse_apply_binary:NwNwN
          \exp_after:wN #2
          \exp:w
          \__fp_parse_operand:Nw #4
          \exp_after:wN \__fp_parse_expand:w
        \else:
          \exp_after:wN @
          \exp_after:wN \use_none:n
          \exp_after:wN #1
        \fi:
      }
  }
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_^:N }   ^
  \c__fp_prec_hatii_int \c__fp_prec_hat_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_juxt:N } *
  \c__fp_prec_juxt_int \c__fp_prec_juxt_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_/:N }   /
  \c__fp_prec_times_int \c__fp_prec_times_int
```

```
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_mul:N } *
  \c__fp_prec_times_int \c__fp_prec_times_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_-:N }   -
  \c__fp_prec_plus_int  \c__fp_prec_plus_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_+:N }   +
  \c__fp_prec_plus_int  \c__fp_prec_plus_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_and:N } &
  \c__fp_prec_and_int   \c__fp_prec_and_int
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_or:N }  |
  \c__fp_prec_or_int    \c__fp_prec_or_int
```

(*End of definition for* `\__fp_parse_infix_+:N` *and others.*)

71.7.3 Juxtaposition

`\__fp_parse_infix_(:N`

When an opening parenthesis appears where we expect an infix operator, we compute the product of the previous operand and the contents of the parentheses using `\__fp_parse_infix_mul:N`.

```
\cs_new:cpn { __fp_parse_infix_(:N } #1
  { \__fp_parse_infix_mul:N #1 ( }
```

(*End of definition for* `\__fp_parse_infix_(:N`.)

71.7.4 Multi-character cases

`\__fp_parse_infix_*:N`

```
\cs_set_protected:Npn \__fp_tmp:w #1
  {
    \cs_new:cpn { __fp_parse_infix_*:N } ##1##2
      {
        \if:w * \exp_not:N ##2
          \exp_after:wN #1
          \exp_after:wN ##1
        \else:
          \exp_after:wN \__fp_parse_infix_mul:N
          \exp_after:wN ##1
          \exp_after:wN ##2
        \fi:
      }
  }
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_^:N }
```

(*End of definition for* `\__fp_parse_infix_*:N`.)

`\__fp_parse_infix_|:Nw`
`\__fp_parse_infix_&:Nw`

```
\cs_set_protected:Npn \__fp_tmp:w #1#2#3
  {
    \cs_new:Npn #1 ##1##2
      {
        \if:w #2 \exp_not:N ##2
          \exp_after:wN #1
          \exp_after:wN ##1
          \exp:w \exp_after:wN \__fp_parse_expand:w
        \else:
```

```
          \exp_after:wN #3
          \exp_after:wN ##1
          \exp_after:wN ##2
        \fi:
      }
  }
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_|:N } | \__fp_parse_infix_or:N
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_&:N } & \__fp_parse_infix_and:N
```

(*End of definition for* `\__fp_parse_infix_|:Nw` *and* `\__fp_parse_infix_&:Nw`.)

71.7.5 Ternary operator

`\__fp_parse_infix_?:N`
`\__fp_parse_infix_::N`

```
\cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
  {
    \cs_new:Npn #1 ##1
      {
        \if_int_compare:w ##1 < \c__fp_prec_quest_int
          #4
          \exp_after:wN @
          \exp_after:wN #2
          \exp:w
          \__fp_parse_operand:Nw #3
          \exp_after:wN \__fp_parse_expand:w
        \else:
          \exp_after:wN @
          \exp_after:wN \use_none:n
          \exp_after:wN #1
        \fi:
      }
  }
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_?:N }
  \__fp_ternary:NwwN \c__fp_prec_quest_int { }
\exp_args:Nc \__fp_tmp:w { __fp_parse_infix_::N }
  \__fp_ternary_auxii:NwwN \c__fp_prec_colon_int
  {
    \msg_expandable_error:nnnn
      { fp } { missing } { ? } { ~for~?: }
  }
```

(*End of definition for* `\__fp_parse_infix_?:N` *and* `\__fp_parse_infix_::N`.)

71.7.6 Comparisons

`\__fp_parse_infix_<:N`
`\__fp_parse_infix_=:N`
`\__fp_parse_infix_>:N`
`\__fp_parse_infix_!:N`
`\__fp_parse_excl_error:`
`\__fp_parse_compare:NNNNNNN`
`\__fp_parse_compare_auxi:NNNNNNN`
`\__fp_parse_compare_auxii:NNNNN`
`\__fp_parse_compare_end:NNNNw`
`\__fp_compare:wNNNNw`

```
\cs_new:cpn { __fp_parse_infix_<:N } #1
  { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 < }
\cs_new:cpn { __fp_parse_infix_=:N } #1
  { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 = }
\cs_new:cpn { __fp_parse_infix_>:N } #1
  { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 > }
\cs_new:cpn { __fp_parse_infix_!:N } #1
  {
```

```
    \exp_after:wN \__fp_parse_compare:NNNNNNN
    \exp_after:wN #1
    \exp_after:wN 0
    \exp_after:wN 1
    \exp_after:wN 1
    \exp_after:wN 1
    \exp_after:wN 1
  }
\cs_new:Npn \__fp_parse_excl_error:
  {
    \msg_expandable_error:nnnn
      { fp } { missing } { = } { ~after~!. }
  }
\cs_new:Npn \__fp_parse_compare:NNNNNNN #1
  {
    \if_int_compare:w #1 < \c__fp_prec_comp_int
      \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
      \exp_after:wN \__fp_parse_excl_error:
    \else:
      \exp_after:wN @
      \exp_after:wN \use_none:n
      \exp_after:wN \__fp_parse_compare:NNNNNNN
    \fi:
  }
\cs_new:Npn \__fp_parse_compare_auxi:NNNNNNN #1#2#3#4#5#6#7
  {
    \if_case:w
      \__fp_int_eval:w \exp_after:wN ` \token_to_str:N #7 - `<
        \__fp_int_eval_end:
         \__fp_parse_compare_auxii:NNNNN #2#2#4#5#6
    \or: \__fp_parse_compare_auxii:NNNNN #2#3#2#5#6
    \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#2#6
    \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#5#2
    \else: #1 \__fp_parse_compare_end:NNNNw #3#4#5#6#7
    \fi:
  }
\cs_new:Npn \__fp_parse_compare_auxii:NNNNN #1#2#3#4#5
  {
    \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
    \exp_after:wN \prg_do_nothing:
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN #4
    \exp_after:wN #5
    \exp:w \exp_after:wN \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
  {
    \fi:
    \exp_after:wN @
    \exp_after:wN \__fp_parse_apply_compare:NwNNNNNwN
    \exp_after:wN \c_one_fp
    \exp_after:wN #1
```

```
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN #4
    \exp:w
    \__fp_parse_operand:Nw \c__fp_prec_comp_int \__fp_parse_expand:w #5
  }
\cs_new:Npn \__fp_parse_apply_compare:NwNNNNNwN
    #1 #2@ #3 #4#5#6#7 #8@ #9
  {
    \if_int_odd:w
        \if_meaning:w \c_zero_fp #3
          0
        \else:
          \if_case:w \__fp_compare_back_any:ww #8 #2 \exp_stop_f:
            #5 \or: #6 \or: #7 \else: #4
          \fi:
        \fi:
        \exp_stop_f:
      \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
      \exp_after:wN \c_one_fp
    \else:
      \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
      \exp_after:wN \c_zero_fp
    \fi:
    #1 #8 #9
  }
\cs_new:Npn \__fp_parse_apply_compare_aux:NNwN #1 #2 #3; #4
  {
    \if_meaning:w \__fp_parse_compare:NNNNNNN #4
      \exp_after:wN \__fp_parse_continue_compare:NNwNN
      \exp_after:wN #1
      \exp_after:wN #2
      \exp:w \exp_end_continue_f:w
      \__fp_exp_after_o:w #3;
      \exp:w \exp_end_continue_f:w
    \else:
      \exp_after:wN \__fp_parse_continue:NwN
      \exp_after:wN #2
      \exp:w \exp_end_continue_f:w
      \exp_after:wN #1
      \exp:w \exp_end_continue_f:w
    \fi:
    #4 #2
  }
\cs_new:Npn \__fp_parse_continue_compare:NNwNN #1#2 #3@ #4#5
  { #4 #2 #3@ #1 }
```

(*End of definition for* `\__fp_parse_infix_<:N` *and others.*)

71.8 Tools for functions

`\__fp_parse_function_all_fp_o:fnw` Followed by {⟨*function name*⟩} {⟨*code*⟩} ⟨`float array`⟩ @ this checks all floats are floating point numbers (no tuples).

```
\cs_new:Npn \__fp_parse_function_all_fp_o:fnw #1#2#3 @
  {
    \__fp_array_if_all_fp:nTF {#3}
      { #2 #3 @ }
      {
        \__fp_error:nffn { bad-args }
          {#1}
          { \fp_to_tl:n { \s__fp_tuple \__fp_tuple_chk:w {#3} ; } }
          { }
        \exp_after:wN \c_nan_fp
      }
  }
```

(*End of definition for* `\__fp_parse_function_all_fp_o:fnw`.)

`\__fp_parse_function_one_two:nnw`
`\__fp_parse_function_one_two_error_o:w`
`\__fp_parse_function_one_two_aux:nnw`
`\__fp_parse_function_one_two_auxii:nnw`

This is followed by `{`⟨*function name*⟩`}` `{`⟨*code*⟩`}` ⟨*float array*⟩ `@`. It checks that the ⟨*float array*⟩ consists of one or two floating point numbers (not tuples), then leaves the ⟨*code*⟩ (if there is one float) or its tail (if there are two floats) followed by the ⟨*float array*⟩. The ⟨*code*⟩ should start with a single token such as `\__fp_atan_default:w` that deals with the single-float case.

The first `\__fp_if_type_fp:NTwFw` test catches the case of no argument and the case of a tuple argument. The next one distinguishes the case of a single argument (no error, just add `\c_one_fp`) from a tuple second argument. Finally check there is no further argument.

```
\cs_new:Npn \__fp_parse_function_one_two:nnw #1#2#3
  {
    \__fp_if_type_fp:NTwFw
      #3 { } \s__fp \__fp_parse_function_one_two_error_o:w \s__fp_stop
    \__fp_parse_function_one_two_aux:nnw {#1} {#2} #3
  }
\cs_new:Npn \__fp_parse_function_one_two_error_o:w #1#2#3#4 @
  {
    \__fp_error:nffn { bad-args }
      {#2}
      { \fp_to_tl:n { \s__fp_tuple \__fp_tuple_chk:w {#4} ; } }
      { }
    \exp_after:wN \c_nan_fp
  }
\cs_new:Npn \__fp_parse_function_one_two_aux:nnw #1#2 #3; #4
  {
    \__fp_if_type_fp:NTwFw
      #4 { }
      \s__fp
      {
        \if_meaning:w @ #4
          \exp_after:wN \use_iv:nnnn
        \fi:
        \__fp_parse_function_one_two_error_o:w
      }
      \s__fp_stop
    \__fp_parse_function_one_two_auxii:nnw {#1} {#2} #3; #4
  }
\cs_new:Npn \__fp_parse_function_one_two_auxii:nnw #1#2#3; #4; #5
  {
```

```
    \if_meaning:w @ #5 \else:
      \exp_after:wN \__fp_parse_function_one_two_error_o:w
    \fi:
    \use_ii:nn {#1} { \use_none:n #2 } #3; #4; #5
  }
```

(*End of definition for* `\__fp_parse_function_one_two:nnw` *and others.*)

`\__fp_tuple_map_o:nw`
`\__fp_tuple_map_loop_o:nw`

Apply `#1` to all items in the following tuple and expand once afterwards. The code `#1` should itself expand once after its result.

```
\cs_new:Npn \__fp_tuple_map_o:nw #1 \s__fp_tuple \__fp_tuple_chk:w #2 ;
  {
    \exp_after:wN \s__fp_tuple
    \exp_after:wN \__fp_tuple_chk:w
    \exp_after:wN {
      \exp:w \exp_end_continue_f:w
      \__fp_tuple_map_loop_o:nw {#1} #2
        { \s__fp \prg_break: } ;
      \prg_break_point:
    \exp_after:wN } \exp_after:wN ;
  }
\cs_new:Npn \__fp_tuple_map_loop_o:nw #1#2#3 ;
  {
    \use_none:n #2
    #1 #2 #3 ;
    \exp:w \exp_end_continue_f:w
    \__fp_tuple_map_loop_o:nw {#1}
  }
```

(*End of definition for* `\__fp_tuple_map_o:nw` *and* `\__fp_tuple_map_loop_o:nw`.)

`\__fp_tuple_mapthread_o:nww`
`\__fp_tuple_mapthread_loop_o:nw`

Apply `#1` to pairs of items in the two following tuples and expand once afterwards.

```
\cs_new:Npn \__fp_tuple_mapthread_o:nww #1
    \s__fp_tuple \__fp_tuple_chk:w #2 ;
    \s__fp_tuple \__fp_tuple_chk:w #3 ;
  {
    \exp_after:wN \s__fp_tuple
    \exp_after:wN \__fp_tuple_chk:w
    \exp_after:wN {
      \exp:w \exp_end_continue_f:w
      \__fp_tuple_mapthread_loop_o:nw {#1}
        #2 { \s__fp \prg_break: } ; @
        #3 { \s__fp \prg_break: } ;
      \prg_break_point:
    \exp_after:wN } \exp_after:wN ;
  }
\cs_new:Npn \__fp_tuple_mapthread_loop_o:nw #1#2#3 ; #4 @ #5#6 ;
  {
    \use_none:n #2
    \use_none:n #5
    #1 #2 #3 ; #5 #6 ;
    \exp:w \exp_end_continue_f:w
    \__fp_tuple_mapthread_loop_o:nw {#1} #4 @
  }
```

(*End of definition for* `\__fp_tuple_mapthread_o:nww` *and* `\__fp_tuple_mapthread_loop_o:nw`.)

71.9 Messages

```
\msg_new:nnn { fp } { deprecated }
  { '#1'~deprecated;~use~'#2' }
\msg_new:nnn { fp } { unknown-fp-word }
  { Unknown~fp~word~#1. }
\msg_new:nnn { fp } { missing }
  { Missing~#1~inserted #2. }
\msg_new:nnn { fp } { extra }
  { Extra~#1~ignored. }
\msg_new:nnn { fp } { early-end }
  { Premature~end~in~fp~expression. }
\msg_new:nnn { fp } { after-e }
  { Cannot~use~#1 after~'e'. }
\msg_new:nnn { fp } { missing-number }
  { Missing~number~before~'#1'. }
\msg_new:nnn { fp } { unknown-symbol }
  { Unknown~symbol~#1~ignored. }
\msg_new:nnn { fp } { extra-comma }
  { Unexpected~comma~turned~to~nan~result. }
\msg_new:nnn { fp } { no-arg }
  { #1~got~no~argument;~used~nan. }
\msg_new:nnn { fp } { multi-arg }
  { #1~got~more~than~one~argument;~used~nan. }
\msg_new:nnn { fp } { num-args }
  { #1~expects~between~#2~and~#3~arguments. }
\msg_new:nnn { fp } { bad-args }
  { Arguments~in~#1#2~are~invalid. }
\msg_new:nnn { fp } { infty-pi }
  { Math~command~#1 is~not~an~fp }
\cs_if_exist:cT { @unexpandable@protect }
  {
    \msg_new:nnn { fp } { robust-cmd }
      { Robust~command~#1 invalid~in~fp~expression! }
  }
```

⟨/package⟩

Chapter 72

l3fp-assign implementation

```
⟨*package⟩
⟨@@=fp⟩
```

72.1 Assigning values

\fp_new:N

Floating point variables are initialized to be +0.

```
\cs_new_protected:Npn \fp_new:N #1
  { \cs_new_eq:NN #1 \c_zero_fp }
\cs_generate_variant:Nn \fp_new:N {c}
```

(*End of definition for* `\fp_new:N`. *This function is documented on page* 259.)

\fp_set:Nn
\fp_set:cn
\fp_set:NV
\fp_set:cV
\fp_gset:Nn
\fp_gset:cn
\fp_gset:NV
\fp_gset:cV
\fp_const:Nn
\fp_const:cn

Simply use `\__fp_parse:n` within various `f`-expanding assignments.

```
\cs_new_protected:Npn \fp_set:Nn   #1#2
  { \__kernel_tl_set:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\cs_new_protected:Npn \fp_gset:Nn  #1#2
  { \__kernel_tl_gset:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\cs_new_protected:Npn \fp_const:Nn #1#2
  { \tl_const:Ne #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\cs_generate_variant:Nn \fp_set:Nn { NV , c , cV }
\cs_generate_variant:Nn \fp_gset:Nn { NV , c , cV }
\cs_generate_variant:Nn \fp_const:Nn {c}
```

(*End of definition for* `\fp_set:Nn`, `\fp_gset:Nn`, *and* `\fp_const:Nn`. *These functions are documented on page* 259.)

\fp_set_eq:NN
\fp_set_eq:cN
\fp_set_eq:Nc
\fp_set_eq:cc
\fp_gset_eq:NN
\fp_gset_eq:cN
\fp_gset_eq:Nc
\fp_gset_eq:cc

Copying a floating point is the same as copying the underlying token list.

```
\cs_new_eq:NN \fp_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
\cs_generate_variant:Nn \fp_set_eq:NN  { c , Nc , cc }
\cs_generate_variant:Nn \fp_gset_eq:NN { c , Nc , cc }
```

(*End of definition for* `\fp_set_eq:NN` *and* `\fp_gset_eq:NN`. *These functions are documented on page* 259.)

\fp_zero:N
\fp_zero:c
\fp_gzero:N
\fp_gzero:c

Setting a floating point to zero: copy \c_zero_fp.

```
\cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }
\cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }
\cs_generate_variant:Nn \fp_zero:N  { c }
\cs_generate_variant:Nn \fp_gzero:N { c }
```

(*End of definition for* \fp_zero:N *and* \fp_gzero:N*. These functions are documented on page 259.*)

\fp_zero_new:N
\fp_zero_new:c
\fp_gzero_new:N
\fp_gzero_new:c

Set the floating point to zero, or define it if needed.

```
\cs_new_protected:Npn \fp_zero_new:N #1
  { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
\cs_new_protected:Npn \fp_gzero_new:N #1
  { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
\cs_generate_variant:Nn \fp_zero_new:N  { c }
\cs_generate_variant:Nn \fp_gzero_new:N { c }
```

(*End of definition for* \fp_zero_new:N *and* \fp_gzero_new:N*. These functions are documented on page 259.*)

72.2 Updating values

These match the equivalent functions in l3int and l3skip.

\fp_add:Nn
\fp_add:cn
\fp_gadd:Nn
\fp_gadd:cn
\fp_sub:Nn
\fp_sub:cn
\fp_gsub:Nn
\fp_gsub:cn
__fp_add:NNNn

For the sake of error recovery we should not simply set #1 to #1 ± (#2): for instance, if #2 is 0)+2, the parsing error would be raised at the last closing parenthesis rather than at the closing parenthesis in the user argument. Thus we evaluate #2 instead of just putting parentheses. As an optimization we use __fp_parse:n rather than \fp_eval:n, which would convert the result away from the internal representation and back.

```
\cs_new_protected:Npn \fp_add:Nn  { \__fp_add:NNNn \fp_set:Nn  + }
\cs_new_protected:Npn \fp_gadd:Nn { \__fp_add:NNNn \fp_gset:Nn + }
\cs_new_protected:Npn \fp_sub:Nn  { \__fp_add:NNNn \fp_set:Nn  - }
\cs_new_protected:Npn \fp_gsub:Nn { \__fp_add:NNNn \fp_gset:Nn - }
\cs_new_protected:Npn \__fp_add:NNNn #1#2#3#4
  { #1 #3 { #3 #2 \__fp_parse:n {#4} } }
\cs_generate_variant:Nn \fp_add:Nn  { c }
\cs_generate_variant:Nn \fp_gadd:Nn { c }
\cs_generate_variant:Nn \fp_sub:Nn  { c }
\cs_generate_variant:Nn \fp_gsub:Nn { c }
```

(*End of definition for* \fp_add:Nn *and others. These functions are documented on page 259.*)

72.3 Showing values

\fp_show:N
\fp_show:c
\fp_log:N
\fp_log:c
__fp_show:NN

This shows the result of computing its argument by passing the right data to \tl_show:n or \tl_log:n.

```
\cs_new_protected:Npn \fp_show:N { \__fp_show:NN \tl_show:n }
\cs_generate_variant:Nn \fp_show:N { c }
\cs_new_protected:Npn \fp_log:N { \__fp_show:NN \tl_log:n }
\cs_generate_variant:Nn \fp_log:N { c }
\cs_new_protected:Npn \__fp_show:NN #1#2
  {
    \__kernel_chk_tl_type:NnnT #2 { fp }
```

```
      { \exp_args:No \__fp_show_validate:n #2 }
      { \exp_args:Ne #1 { \token_to_str:N #2 = \fp_to_tl:N #2 } }
  }
```

(*End of definition for* `\fp_show:N`*,* `\fp_log:N`*, and* `\__fp_show:NN`*. These functions are documented on page 270.*)

`\__fp_show_validate:n`
`\__fp_show_validate_aux:n`
`\__fp_show_validate:nn`
`\__fp_show_validate:w`
`\__fp_tuple_show_validate:w`
`\__fp_symbolic_show_validate:w`

To support symbolic expression, validation has to be done recursively. Two `\@@_show_validate:nn` wrappers are used to distinguish between initial and recursive calls, in which the former provides a demo of possible forms a `fp` variable would have.

```
\cs_new:Npn \__fp_show_validate:n #1
  {
    \__fp_show_validate:nn { #1 }
      {
        \s__fp \__fp_chk:w ??? ;~ or \iow_newline:
        \s__fp_tuple \__fp_tuple_chk:w ? ;~ or \iow_newline:
        \s__fp_symbolic \__fp_symbolic_chk:w ? , ? ;
      }
  }
\cs_new:Npn \__fp_show_validate_aux:n #1
  {
    \__fp_show_validate:nn { #1 } { }
  }
\cs_new:Npn \__fp_show_validate:nn #1#2
  {
    \tl_if_empty:nF { #1 }
      {
        \str_case:enF { \tl_head:n { #1 } }
          {
            { \s__fp }
              {
                \__fp_show_validate:w #1 \s__fp
                \__fp_chk:w ??? ; \s__fp_stop
              }
            { \s__fp_tuple }
              {
                \__fp_tuple_show_validate:w #1
                \s__fp_tuple \__fp_tuple_chk:w ?? ; \s__fp_stop
              }
            { \s__fp_symbolic }
              {
                \__fp_symbolic_show_validate:w #1
                \s__fp_symbolic \__fp_symbolic_chk:w ? , ?? ; \s__fp_stop
              }
          }
          { #2 }
      }
  }
\cs_new:Npn \__fp_show_validate:w
    #1 \s__fp \__fp_chk:w #2#3#4#5 ; #6 \s__fp_stop
  {
    \str_if_eq:nnF { #2 } {?}
      {
        \token_if_eq_meaning:NNTF #2 1
```

```
            { \s__fp \__fp_chk:w #2 #3 { #4 } #5 ; }
            { \s__fp \__fp_chk:w #2 #3 #4 #5 ; }
          \__fp_show_validate_aux:n { #6 }
        }
    }
\cs_new:Npn \__fp_tuple_show_validate:w
    #1 \s__fp_tuple \__fp_tuple_chk:w #2#3 ; #4 \s__fp_stop
  {
    \str_if_eq:nnF { #2 } {?}
      { \s__fp_tuple \__fp_tuple_chk:w { \__fp_show_validate_aux:n { #2 } } ; }
  }
\cs_new:Npn \__fp_symbolic_show_validate:w
    #1 \s__fp_symbolic \__fp_symbolic_chk:w #2 , #3#4 ; #5 \s__fp_stop
  {
    \str_if_eq:nnF { #2 } {?}
      {
        \s__fp_symbolic \__fp_symbolic_chk:w \exp_not:n { #2 } ,
        { \__fp_show_validate_aux:n { #3 } };
        \__fp_show_validate_aux:n { #5 }
      }
  }
```

(*End of definition for* `\__fp_show_validate:n` *and others.*)

`\fp_show:n`
`\fp_log:n`

Use general tools.

```
\cs_new_protected:Npn \fp_show:n
  { \__kernel_msg_show_eval:Nn \fp_to_tl:n }
\cs_new_protected:Npn \fp_log:n
  { \__kernel_msg_log_eval:Nn \fp_to_tl:n }
```

(*End of definition for* `\fp_show:n` *and* `\fp_log:n`. *These functions are documented on page 270.*)

72.4 Some useful constants and scratch variables

`\c_one_fp`
`\c_e_fp`

Some constants.

```
\fp_const:Nn \c_e_fp          { 2.718 2818 2845 9045 }
\fp_const:Nn \c_one_fp        { 1 }
```

(*End of definition for* `\c_one_fp` *and* `\c_e_fp`. *These variables are documented on page 268.*)

`\c_pi_fp`
`\c_one_degree_fp`

We simply round π to and $\pi/180$ to 16 significant digits.

```
\fp_const:Nn \c_pi_fp         { 3.141 5926 5358 9793 }
\fp_const:Nn \c_one_degree_fp { 0.0 1745 3292 5199 4330 }
```

(*End of definition for* `\c_pi_fp` *and* `\c_one_degree_fp`. *These variables are documented on page 268.*)

`\l_tmpa_fp`
`\l_tmpb_fp`
`\g_tmpa_fp`
`\g_tmpb_fp`

Scratch variables are simply initialized there.

```
\fp_new:N \l_tmpa_fp
\fp_new:N \l_tmpb_fp
\fp_new:N \g_tmpa_fp
\fp_new:N \g_tmpb_fp
```

(*End of definition for* `\l_tmpa_fp` *and others. These variables are documented on page 268.*)

```
⟨/package⟩
```

Chapter 73

l3fp-logic implementation

```
⟨*package⟩
⟨@@=fp⟩
```

__fp_parse_word_max:N
__fp_parse_word_min:N

Those functions may receive a variable number of arguments.

```
\cs_new:Npn \__fp_parse_word_max:N
  { \__fp_parse_function:NNN \__fp_minmax_o:Nw 2 }
\cs_new:Npn \__fp_parse_word_min:N
  { \__fp_parse_function:NNN \__fp_minmax_o:Nw 0 }
```

(*End of definition for* __fp_parse_word_max:N *and* __fp_parse_word_min:N.)

73.1 Syntax of internal functions

- __fp_compare_npos:nwnw {⟨*expo*$_1$⟩} ⟨*body*$_1$⟩ ; {⟨*expo*$_2$⟩} ⟨*body*$_2$⟩ ;
- __fp_minmax_o:Nw ⟨*sign*⟩ ⟨*floating point array*⟩
- __fp_not_o:w ? ⟨*floating point array*⟩ (with one floating point number only)
- __fp_&_o:ww ⟨*floating point*⟩ ⟨*floating point*⟩
- __fp_|_o:ww ⟨*floating point*⟩ ⟨*floating point*⟩
- __fp_ternary:NwwN, __fp_ternary_auxi:NwwN, __fp_ternary_auxii:NwwN have to be understood.

73.2 Tests

\fp_if_exist_p:N
\fp_if_exist_p:c
\fp_if_exist:NTF
\fp_if_exist:cTF

Copies of the cs functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \fp_if_exist:N \cs_if_exist:N { TF , T , F , p }
\prg_new_eq_conditional:NNn \fp_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(*End of definition for* \fp_if_exist:NTF. *This function is documented on page* *261*.)

\fp_if_nan_p:n
\fp_if_nan:nTF

Evaluate and check if the result is a floating point of the same kind as nan.

```
\prg_new_conditional:Npnn \fp_if_nan:n #1 { TF , T , F , p }
  {
    \if:w 3 \exp_last_unbraced:Nf \__fp_kind:w { \__fp_parse:n {#1} }
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End of definition for* \fp_if_nan:nTF. *This function is documented on page* 263.)

73.3 Comparison

\fp_compare_p:n
\fp_compare:nTF
__fp_compare_return:w

Within floating point expressions, comparison operators are treated as operations, so we evaluate #1, then compare with ± 0. Tuples are true.

```
\prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }
  {
    \exp_after:wN \__fp_compare_return:w
    \exp:w \exp_end_continue_f:w \__fp_parse:n {#1}
  }
\cs_new:Npn \__fp_compare_return:w #1#2#3;
  {
    \if_charcode:w 0
          \__fp_if_type_fp:NTwFw
            #1 { \__fp_use_i_delimit_by_s_stop:nw #3 \s__fp_stop }
            \s__fp 1 \s__fp_stop
      \prg_return_false:
    \else:
      \prg_return_true:
    \fi:
  }
```

(*End of definition for* \fp_compare:nTF *and* __fp_compare_return:w. *This function is documented on page* 263.)

\fp_compare_p:nNn
\fp_compare:nNnTF
__fp_compare_aux:wn

Evaluate #1 and #3, using an auxiliary to expand both, and feed the two floating point numbers swapped to __fp_compare_back_any:ww, defined below. Compare the result with ‘#2-‘=, which is -1 for <, 0 for =, 1 for > and 2 for ?.

```
\prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
  {
    \if_int_compare:w
        \exp_after:wN \__fp_compare_aux:wn
          \exp:w \exp_end_continue_f:w \__fp_parse:n {#1} {#3}
        = \__fp_int_eval:w ‘#2 - ‘= \__fp_int_eval_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \__fp_compare_aux:wn #1; #2
  {
    \exp_after:wN \__fp_compare_back_any:ww
```

```
      \exp:w \exp_end_continue_f:w \__fp_parse:n {#2} #1;
  }
```

(*End of definition for* `\fp_compare:nNnTF` *and* `\__fp_compare_aux:wn`*. This function is documented on page* 262*.*)

`\__fp_compare_back:ww`
`\__fp_bcmp:ww`
`\__fp_compare_back_any:ww`
`\__fp_compare_nan:w`

`\__fp_compare_back_any:ww` ⟨y⟩ `;` ⟨x⟩ `;`

Expands (in the same way as `\int_eval:n`) to -1 if $x < y$, 0 if $x = y$, 1 if $x > y$, and 2 otherwise (denoted as $x?y$). If either operand is `nan`, stop the comparison with `\__fp_compare_nan:w` returning 2. If x is negative, swap the outputs 1 and -1 (*i.e.*, $>$ and $<$); we can henceforth assume that $x \geq 0$. If $y \geq 0$, and they have the same type, either they are normal and we compare them with `\__fp_compare_npos:nwnw`, or they are equal. If $y \geq 0$, but of a different type, the highest type is a larger number. Finally, if $y \leq 0$, then $x > y$, unless both are zero.

```
\cs_new:Npn \__fp_compare_back:ww #1#2; #3#4;
  {
    \cs:w
      __fp
      \__fp_type_from_scan:N #1
      _bcmp
      \__fp_type_from_scan:N #3
      :ww
    \cs_end:
    #1#2; #3#4;
  }
\cs_new:Npn \__fp_compare_back_any:ww #1#2; #3
  {
    \__fp_if_type_fp:NTwFw
      #1 { \__fp_if_type_fp:NTwFw #3 \use_i:nn \s__fp \use_ii:nn \s__fp_stop }
      \s__fp \use_ii:nn \s__fp_stop
    \__fp_compare_back:ww
    {
      \cs:w
        __fp
        \__fp_type_from_scan:N #1
        _compare_back
        \__fp_type_from_scan:N #3
        :ww
      \cs_end:
    }
    #1#2 ; #3
  }
\cs_new:Npn \__fp_bcmp:ww
    \s__fp \__fp_chk:w #1 #2 #3;
    \s__fp \__fp_chk:w #4 #5 #6;
  {
    \int_value:w
      \if_meaning:w 3 #1 \exp_after:wN \__fp_compare_nan:w \fi:
      \if_meaning:w 3 #4 \exp_after:wN \__fp_compare_nan:w \fi:
      \if_meaning:w 2 #5 - \fi:
      \if_meaning:w #2 #5
        \if_meaning:w #1 #4
          \if_meaning:w 1 #1
            \__fp_compare_npos:nwnw #6; #3;
```

```
          \else:
            0
          \fi:
        \else:
          \if_int_compare:w #4 < #1 - \fi: 1
        \fi:
      \else:
        \if_int_compare:w #1#4 = \c_zero_int
          0
        \else:
          1
        \fi:
      \fi:
    \exp_stop_f:
  }
\cs_new:Npn \__fp_compare_nan:w #1 \fi: \exp_stop_f: { 2 \exp_stop_f: }
```

(*End of definition for* `\__fp_compare_back:ww` *and others.*)

`\__fp_compare_back_tuple:ww`
`\__fp_tuple_compare_back:ww`
`\__fp_tuple_compare_back_tuple:ww`
`\__fp_tuple_compare_back_loop:w`

Tuple and floating point numbers are not comparable so return 2 in mixed cases or when tuples have a different number of items. Otherwise compare pairs of items with `\__fp_compare_back_any:ww` and if any don't match return 2 (as `\int_value:w 02 \exp_stop_f:`).

```
\cs_new:Npn \__fp_compare_back_tuple:ww #1; #2; { 2 }
\cs_new:Npn \__fp_tuple_compare_back:ww #1; #2; { 2 }
\cs_new:Npn \__fp_tuple_compare_back_tuple:ww
  \s__fp_tuple \__fp_tuple_chk:w #1;
  \s__fp_tuple \__fp_tuple_chk:w #2;
  {
    \int_compare:nNnTF { \__fp_array_count:n {#1} } =
      { \__fp_array_count:n {#2} }
      {
        \int_value:w 0
          \__fp_tuple_compare_back_loop:w
              #1 { \s__fp \prg_break: } ; @
              #2 { \s__fp \prg_break: } ;
            \prg_break_point:
        \exp_stop_f:
      }
      { 2 }
  }
\cs_new:Npn \__fp_tuple_compare_back_loop:w #1#2 ; #3 @ #4#5 ;
  {
    \use_none:n #1
    \use_none:n #4
    \if_int_compare:w
        \__fp_compare_back_any:ww #1 #2 ; #4 #5 ; = \c_zero_int
    \else:
      2 \exp_after:wN \prg_break:
    \fi:
    \__fp_tuple_compare_back_loop:w #3 @
  }
```

(*End of definition for* `\__fp_compare_back_tuple:ww` *and others.*)

__fp_compare_npos:nwnw
__fp_compare_significand:nnnnnnnn

__fp_compare_npos:nwnw {⟨*expo*$_1$⟩} ⟨*body*$_1$⟩ ; {⟨*expo*$_2$⟩} ⟨*body*$_2$⟩ ;

Within an \int_value:w ... \exp_stop_f: construction, this expands to 0 if the two numbers are equal, -1 if the first is smaller, and 1 if the first is bigger. First compare the exponents: the larger one denotes the larger number. If they are equal, we must compare significands. If both the first 8 digits and the next 8 digits coincide, the numbers are equal. If only the first 8 digits coincide, the next 8 decide. Otherwise, the first 8 digits are compared.

```
\cs_new:Npn \__fp_compare_npos:nwnw #1#2; #3#4;
  {
    \if_int_compare:w #1 = #3 \exp_stop_f:
      \__fp_compare_significand:nnnnnnnn #2 #4
    \else:
      \if_int_compare:w #1 < #3 - \fi: 1
    \fi:
  }
\cs_new:Npn \__fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
  {
    \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
      \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
        0
      \else:
        \if_int_compare:w #3#4 < #7#8 - \fi: 1
      \fi:
    \else:
      \if_int_compare:w #1#2 < #5#6 - \fi: 1
    \fi:
  }
```

(*End of definition for* __fp_compare_npos:nwnw *and* __fp_compare_significand:nnnnnnnn.)

73.4 Floating point expression loops

\fp_do_until:nn
\fp_do_while:nn
\fp_until_do:nn
\fp_while_do:nn

These are quite easy given the above functions. The do_until and do_while versions execute the body, then test. The until_do and while_do do it the other way round.

```
\cs_new:Npn \fp_do_until:nn #1#2
  {
    #2
    \fp_compare:nF {#1}
      { \fp_do_until:nn {#1} {#2} }
  }
\cs_new:Npn \fp_do_while:nn #1#2
  {
    #2
    \fp_compare:nT {#1}
      { \fp_do_while:nn {#1} {#2} }
  }
\cs_new:Npn \fp_until_do:nn #1#2
  {
    \fp_compare:nF {#1}
      {
        #2
        \fp_until_do:nn {#1} {#2}
```

```
      }
  }
\cs_new:Npn \fp_while_do:nn #1#2
  {
    \fp_compare:nT {#1}
      {
        #2
        \fp_while_do:nn {#1} {#2}
      }
  }
```

(*End of definition for* `\fp_do_until:nn` *and others. These functions are documented on page 264.*)

`\fp_do_until:nNnn`
`\fp_do_while:nNnn`
`\fp_until_do:nNnn`
`\fp_while_do:nNnn`

As above but not using the `nNn` syntax.

```
\cs_new:Npn \fp_do_until:nNnn #1#2#3#4
  {
    #4
    \fp_compare:nNnF {#1} #2 {#3}
      { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
  }
\cs_new:Npn \fp_do_while:nNnn #1#2#3#4
  {
    #4
    \fp_compare:nNnT {#1} #2 {#3}
      { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
  }
\cs_new:Npn \fp_until_do:nNnn #1#2#3#4
  {
    \fp_compare:nNnF {#1} #2 {#3}
      {
        #4
        \fp_until_do:nNnn {#1} #2 {#3} {#4}
      }
  }
\cs_new:Npn \fp_while_do:nNnn #1#2#3#4
  {
    \fp_compare:nNnT {#1} #2 {#3}
      {
        #4
        \fp_while_do:nNnn {#1} #2 {#3} {#4}
      }
  }
```

(*End of definition for* `\fp_do_until:nNnn` *and others. These functions are documented on page 263.*)

`\fp_step_function:nnnN`
`\fp_step_function:nnnc`
`\__fp_step:wwwN`
`\__fp_step_fp:wwwN`
`\__fp_step:NnnnnN`
`\__fp_step:NfnnnN`

The approach here is somewhat similar to `\int_step_function:nnnN`. There are two subtleties: we use the internal parser `\__fp_parse:n` to avoid converting back and forth from the internal representation; and (due to rounding) even a non-zero step does not guarantee that the loop counter increases.

```
\cs_new:Npn \fp_step_function:nnnN #1#2#3
  {
    \exp_after:wN \__fp_step:wwwN
      \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#1}
      \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#2}
```

```
      \exp:w \exp_end_continue_f:w \__fp_parse:n {#3}
  }
\cs_generate_variant:Nn \fp_step_function:nnnN { nnnc }
```

Only floating point numbers (not tuples) are allowed arguments. Only "normal" floating points (not ±0, ±`inf`, `nan`) can be used as step; if positive, call `\__fp_step:NnnnnN` with argument `>` otherwise `<`. This function has one more argument than its integer counterpart, namely the previous value, to catch the case where the loop has made no progress. Conversion to decimal is done just before calling the user's function.

```
\cs_new:Npn \__fp_step:wwwN #1#2; #3#4; #5#6; #7
  {
    \__fp_if_type_fp:NTwFw #1 { } \s__fp \prg_break: \s__fp_stop
    \__fp_if_type_fp:NTwFw #3 { } \s__fp \prg_break: \s__fp_stop
    \__fp_if_type_fp:NTwFw #5 { } \s__fp \prg_break: \s__fp_stop
    \use_i:nnnn { \__fp_step_fp:wwwN #1#2; #3#4; #5#6; #7 }
    \prg_break_point:
    \use:n
      {
        \__fp_error:nfff { step-tuple } { \fp_to_tl:n { #1#2 ; } }
          { \fp_to_tl:n { #3#4 ; } } { \fp_to_tl:n { #5#6 ; } }
      }
  }
\cs_new:Npn \__fp_step_fp:wwwN #1 ; \s__fp \__fp_chk:w #2#3#4 ; #5; #6
  {
    \token_if_eq_meaning:NNTF #2 1
      {
        \token_if_eq_meaning:NNTF #3 0
          { \__fp_step:NnnnnN > }
          { \__fp_step:NnnnnN < }
      }
      {
        \token_if_eq_meaning:NNTF #2 0
          {
            \msg_expandable_error:nnn { kernel }
              { zero-step } {#6}
          }
          {
            \__fp_error:nnfn { bad-step } { }
              { \fp_to_tl:n { \s__fp \__fp_chk:w #2#3#4 ; } } {#6}
          }
        \use_none:nnnnn
      }
      { #1 ; } { \c_nan_fp } { \s__fp \__fp_chk:w #2#3#4 ; } { #5 ; } #6
  }
\cs_new:Npn \__fp_step:NnnnnN #1#2#3#4#5#6
  {
    \fp_compare:nNnTF {#2} = {#3}
      {
        \__fp_error:nffn { tiny-step }
          { \fp_to_tl:n {#3} } { \fp_to_tl:n {#4} } {#6}
      }
      {
        \fp_compare:nNnF {#2} #1 {#5}
          {
```

```
            \exp_args:Nf #6 { \__fp_to_decimal_dispatch:w #2 }
            \__fp_step:NfnnnN
              #1 { \__fp_parse:n { #2 + #4 } } {#2} {#4} {#5} #6
          }
      }
  }
\cs_generate_variant:Nn \__fp_step:NnnnnN { Nf }
```

(*End of definition for* `\fp_step_function:nnnN` *and others. This function is documented on page* 265.)

`\fp_step_inline:nnnn`
`\fp_step_variable:nnnNn`
`\__fp_step:NNnnnn`

As for `\int_step_inline:nnnn`, create a global function and apply it, following up with a break point.

```
\cs_new_protected:Npn \fp_step_inline:nnnn
  {
    \int_gincr:N \g__kernel_prg_map_int
    \exp_args:NNc \__fp_step:NNnnnn
      \cs_gset_protected:Npn
      { __fp_map_ \int_use:N \g__kernel_prg_map_int :w }
  }
\cs_new_protected:Npn \fp_step_variable:nnnNn #1#2#3#4#5
  {
    \int_gincr:N \g__kernel_prg_map_int
    \exp_args:NNc \__fp_step:NNnnnn
      \cs_gset_protected:Npe
      { __fp_map_ \int_use:N \g__kernel_prg_map_int :w }
      {#1} {#2} {#3}
      {
        \tl_set:Nn \exp_not:N #4 {##1}
        \exp_not:n {#5}
      }
  }
\cs_new_protected:Npn \__fp_step:NNnnnn #1#2#3#4#5#6
  {
    #1 #2 ##1 {#6}
    \fp_step_function:nnnN {#3} {#4} {#5} #2
    \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
  }
```

(*End of definition for* `\fp_step_inline:nnnn`*,* `\fp_step_variable:nnnNn`*, and* `\__fp_step:NNnnnn`*. These functions are documented on page* 265.)

```
\msg_new:nnn { fp } { step-tuple }
  { Tuple~argument~in~fp_step_...~{#1}{#2}{#3}. }
\msg_new:nnn { fp } { bad-step }
  { Invalid~step~size~#2~for~function~#3. }
\msg_new:nnn { fp } { tiny-step }
  { Tiny~step~size~(#1+#2=#1)~for~function~#3. }
```

73.5 Extrema

`\__fp_minmax_o:Nw`
`\__fp_minmax_aux_o:Nw`

First check all operands are floating point numbers. The argument `#1` is 2 to find the maximum of an array `#2` of floating point numbers, and 0 to find the minimum. We read numbers sequentially, keeping track of the largest (smallest) number found so far. If numbers are equal (for instance ± 0), the first is kept. We append $-\infty$ (∞), for the case of an empty array. Since no number is smaller (larger) than that, this additional item only affects the maximum (minimum) in the case of `max()` and `min()` with no argument. The weird fp-like trailing marker breaks the loop correctly: see the precise definition of `\__fp_minmax_loop:Nww`.

```
\cs_new:Npn \__fp_minmax_o:Nw #1
  {
    \__fp_parse_function_all_fp_o:fnw
      { \token_if_eq_meaning:NNTF 0 #1 { min } { max } }
      { \__fp_minmax_aux_o:Nw #1 }
  }
\cs_new:Npn \__fp_minmax_aux_o:Nw #1#2 @
  {
    \if_meaning:w 0 #1
      \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN +
    \else:
      \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN -
    \fi:
    #2
    \s__fp \__fp_chk:w 2 #1 \s__fp_exact ;
    \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;
  }
```

(*End of definition for* `\__fp_minmax_o:Nw` *and* `\__fp_minmax_aux_o:Nw`.)

`\__fp_minmax_loop:Nww` The first argument is − or + to denote the case where the currently largest (smallest) number found (first floating point argument) should be replaced by the new number (second floating point argument). If the new number is `nan`, keep that as the extremum, unless that extremum is already a `nan`. Otherwise, compare the two numbers. If the new number is larger (in the case of `max`) or smaller (in the case of `min`), the test yields `true`, and we keep the second number as a new maximum; otherwise we keep the first number. Then loop.

```
\cs_new:Npn \__fp_minmax_loop:Nww
    #1 \s__fp \__fp_chk:w #2#3; \s__fp \__fp_chk:w #4#5;
  {
    \if_meaning:w 3 #4
      \if_meaning:w 3 #2
        \__fp_minmax_auxi:ww
      \else:
        \__fp_minmax_auxii:ww
      \fi:
    \else:
      \if_int_compare:w
          \__fp_compare_back:ww
            \s__fp \__fp_chk:w #4#5;
            \s__fp \__fp_chk:w #2#3;
          = #1 1 \exp_stop_f:
        \__fp_minmax_auxii:ww
      \else:
        \__fp_minmax_auxi:ww
      \fi:
    \fi:
    \__fp_minmax_loop:Nww #1
      \s__fp \__fp_chk:w #2#3;
```

```
      \s__fp \__fp_chk:w #4#5;
  }
```

(*End of definition for* `\__fp_minmax_loop:Nww`.)

`\__fp_minmax_auxi:ww`
`\__fp_minmax_auxii:ww`

Keep the first/second number, and remove the other.

```
\cs_new:Npn \__fp_minmax_auxi:ww  #1 \fi: \fi: #2 \s__fp #3 ; \s__fp #4;
  { \fi: \fi: #2 \s__fp #3 ; }
\cs_new:Npn \__fp_minmax_auxii:ww #1 \fi: \fi: #2 \s__fp #3 ;
  { \fi: \fi: #2 }
```

(*End of definition for* `\__fp_minmax_auxi:ww` *and* `\__fp_minmax_auxii:ww`.)

`\__fp_minmax_break_o:w`

This function is called from within an `\if_meaning:w` test. Skip to the end of the tests, close the current test with `\fi:`, clean up, and return the appropriate number with one post-expansion.

```
\cs_new:Npn \__fp_minmax_break_o:w #1 \fi: \fi: #2 \s__fp #3; #4;
  { \fi: \__fp_exp_after_o:w \s__fp #3; }
```

(*End of definition for* `\__fp_minmax_break_o:w`.)

73.6 Boolean operations

`\__fp_not_o:w`
`\__fp_tuple_not_o:w`

Return `true` or `false`, with two expansions, one to exit the conditional, and one to please l3fp-parse. The first argument is provided by l3fp-parse and is ignored.

```
\cs_new:Npn \__fp_not_o:w #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_meaning:w 0 #2
      \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
    \else:
      \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
    \fi:
  }
\cs_new:Npn \__fp_tuple_not_o:w #1 @ { \exp_after:wN \c_zero_fp }
```

(*End of definition for* `\__fp_not_o:w` *and* `\__fp_tuple_not_o:w`.)

`\__fp_&_o:ww`
`\__fp_tuple_&_o:ww`
`\__fp_&_tuple_o:ww`
`\__fp_tuple_&_tuple_o:ww`
`\__fp_|_o:ww`
`\__fp_tuple_|_o:ww`
`\__fp_|_tuple_o:ww`
`\__fp_tuple_|_tuple_o:ww`
`\__fp_and_return:wNw`

For `and`, if the first number is zero, return it (with the same sign). Otherwise, return the second one. For `or`, the logic is reversed: if the first number is non-zero, return it, otherwise return the second number: we achieve that by hi-jacking `\__fp_&_o:ww`, inserting an extra argument, `\else:`, before `\s__fp`. In all cases, expand after the floating point number.

```
\group_begin:
  \char_set_catcode_letter:N &
  \char_set_catcode_letter:N |
  \cs_new:Npn \__fp_&_o:ww #1 \s__fp \__fp_chk:w #2#3;
    {
      \if_meaning:w 0 #2 #1
        \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
      \fi:
      \__fp_exp_after_o:w
    }
  \cs_new:Npn \__fp_&_tuple_o:ww #1 \s__fp \__fp_chk:w #2#3;
```

```
    {
      \if_meaning:w 0 #2 #1
        \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
      \fi:
      \__fp_exp_after_tuple_o:w
    }
  \cs_new:Npn \__fp_tuple_&_o:ww #1; { \__fp_exp_after_o:w }
  \cs_new:Npn \__fp_tuple_&_tuple_o:ww #1; { \__fp_exp_after_tuple_o:w }
  \cs_new:Npn \__fp_|_o:ww { \__fp_&_o:ww \else: }
  \cs_new:Npn \__fp_|_tuple_o:ww { \__fp_&_tuple_o:ww \else: }
  \cs_new:Npn \__fp_tuple_|_o:ww #1; #2; { \__fp_exp_after_tuple_o:w #1; }
  \cs_new:Npn \__fp_tuple_|_tuple_o:ww #1; #2;
    { \__fp_exp_after_tuple_o:w #1; }
\group_end:
\cs_new:Npn \__fp_and_return:wNw #1; \fi: #2;
  { \fi: \__fp_exp_after_o:w #1; }
```

(*End of definition for* `\__fp_&_o:ww` *and others.*)

73.7 Ternary operator

`\__fp_ternary:NwwN`
`\__fp_ternary_auxi:NwwN`
`\__fp_ternary_auxii:NwwN`

The first function receives the test and the true branch of the `?:` ternary operator. It calls `\__fp_ternary_auxii:NwwN` if the test branch is a floating point number ± 0, and otherwise calls `\__fp_ternary_auxi:NwwN`. These functions select one of their two arguments.

```
\cs_new:Npn \__fp_ternary:NwwN #1 #2#3@ #4@ #5
  {
    \if_meaning:w \__fp_parse_infix_::N #5
      \if_charcode:w 0
            \__fp_if_type_fp:NTwFw
              #2 { \use_i:nn \__fp_use_i_delimit_by_s_stop:nw #3 \s__fp_stop }
              \s__fp 1 \s__fp_stop
        \exp_after:wN \exp_after:wN \exp_after:wN \__fp_ternary_auxii:NwwN
      \else:
        \exp_after:wN \exp_after:wN \exp_after:wN \__fp_ternary_auxi:NwwN
      \fi:
      \exp_after:wN #1
      \exp:w \exp_end_continue_f:w
      \__fp_exp_after_array_f:w #4 \s__fp_expr_stop
      \exp_after:wN @
      \exp:w
        \__fp_parse_operand:Nw \c__fp_prec_colon_int
        \__fp_parse_expand:w
    \else:
      \msg_expandable_error:nnnn
        { fp } { missing } { : } { ~for~?: }
      \exp_after:wN \__fp_parse_continue:NwN
      \exp_after:wN #1
      \exp:w \exp_end_continue_f:w
      \__fp_exp_after_array_f:w #4 \s__fp_expr_stop
      \exp_after:wN #5
      \exp_after:wN #1
    \fi:
```

```
  }
\cs_new:Npn \__fp_ternary_auxi:NwwN #1#2@#3@#4
  {
    \exp_after:wN \__fp_parse_continue:NwN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
    \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
    #4 #1
  }
\cs_new:Npn \__fp_ternary_auxii:NwwN #1#2@#3@#4
  {
    \exp_after:wN \__fp_parse_continue:NwN
    \exp_after:wN #1
    \exp:w \exp_end_continue_f:w
    \__fp_exp_after_array_f:w #3 \s__fp_expr_stop
    #4 #1
  }
```

(*End of definition for* `\__fp_ternary:NwwN`*,* `\__fp_ternary_auxi:NwwN`*, and* `\__fp_ternary_auxii:NwwN`*.*)

⟨/package⟩

Chapter 74

l3fp-basics implementation

```
⟨*package⟩
⟨@@=fp⟩
```

The l3fp-basics module implements addition, subtraction, multiplication, and division of two floating points, and the absolute value and sign-changing operations on one floating point. All operations implemented in this module yield the outcome of rounding the infinitely precise result of the operation to the nearest floating point.

Some algorithms used below end up being quite similar to some described in "What Every Computer Scientist Should Know About Floating Point Arithmetic", by David Goldberg, which can be found at `http://cr.yp.to/2005-590/goldberg.pdf`.

__fp_parse_word_abs:N
__fp_parse_word_logb:N
__fp_parse_word_sign:N
__fp_parse_word_sqrt:N

Unary functions.

```
\cs_new:Npn \__fp_parse_word_abs:N
  { \__fp_parse_unary_function:NNN \__fp_set_sign_o:w 0 }
\cs_new:Npn \__fp_parse_word_logb:N
  { \__fp_parse_unary_function:NNN \__fp_logb_o:w ? }
\cs_new:Npn \__fp_parse_word_sign:N
  { \__fp_parse_unary_function:NNN \__fp_sign_o:w ? }
\cs_new:Npn \__fp_parse_word_sqrt:N
  { \__fp_parse_unary_function:NNN \__fp_sqrt_o:w ? }
```

(*End of definition for* `\__fp_parse_word_abs:N` *and others.*)

74.1 Addition and subtraction

We define here two functions, `\__fp_-_o:ww` and `\__fp_+_o:ww`, which perform the subtraction and addition of their two floating point operands, and expand the tokens following the result once.

A more obscure function, `\__fp_add_big_i_o:wNww`, is used in l3fp-expo.

The logic goes as follows:

- `\__fp_-_o:ww` calls `\__fp_+_o:ww` to do the work, with the sign of the second operand flipped;
- `\__fp_+_o:ww` dispatches depending on the type of floating point, calling specialized auxiliaries;

- in all cases except summing two normal floating point numbers, we return one or the other operands depending on the signs, or detect an invalid operation in the case of $\infty - \infty$;
- for normal floating point numbers, compare the signs;
- to add two floating point numbers of the same sign or of opposite signs, shift the significand of the smaller one to match the bigger one, perform the addition or subtraction of significands, check for a carry, round, and pack using the `\__fp_basics_pack_...` functions.

The trickiest part is to round correctly when adding or subtracting normal floating point numbers.

74.1.1 Sign, exponent, and special numbers

`\__fp_-_o:ww` The `\__fp_+_o:ww` auxiliary has a hook: it takes one argument between the first `\s__fp` and `\__fp_chk:w`, which is applied to the sign of the second operand. Positioning the hook there means that `\__fp_+_o:ww` can still perform the sanity check that it was followed by `\s__fp`.

```
\cs_new:cpe { __fp_-_o:ww } \s__fp
  {
    \exp_not:c { __fp_+_o:ww }
    \exp_not:n { \s__fp \__fp_neg_sign:N }
  }
```

(*End of definition for* `\__fp_-_o:ww`.)

`\__fp_+_o:ww` This function is either called directly with an empty `#1` to compute an addition, or it is called by `\__fp_-_o:ww` with `\__fp_neg_sign:N` as `#1` to compute a subtraction, in which case the second operand's sign should be changed. If the ⟨*types*⟩ `#2` and `#4` are the same, dispatch to case `#2` (0, 1, 2, or 3), where we call specialized functions: thanks to `\int_value:w`, those receive the tweaked ⟨$sign_2$⟩ (expansion of `#1#5`) as an argument. If the ⟨*types*⟩ are distinct, the result is simply the floating point number with the highest ⟨*type*⟩. Since case 3 (used for two `nan`) also picks the first operand, we can also use it when ⟨$type_1$⟩ is greater than ⟨$type_2$⟩. Also note that we don't need to worry about ⟨$sign_2$⟩ in that case since the second operand is discarded.

```
\cs_new:cpn { __fp_+_o:ww }
    \s__fp #1 \__fp_chk:w #2 #3 ; \s__fp \__fp_chk:w #4 #5
  {
    \if_case:w
      \if_meaning:w #2 #4
        #2
      \else:
        \if_int_compare:w #2 > #4 \exp_stop_f:
          3
        \else:
          4
        \fi:
      \fi:
      \exp_stop_f:
           \exp_after:wN \__fp_add_zeros_o:Nww \int_value:w
    \or:   \exp_after:wN \__fp_add_normal_o:Nww \int_value:w
```

```
    \or:   \exp_after:wN \__fp_add_inf_o:Nww \int_value:w
    \or:   \__fp_case_return_i_o:ww
    \else: \exp_after:wN \__fp_add_return_ii_o:Nww \int_value:w
    \fi:
    #1 #5
    \s__fp \__fp_chk:w #2 #3 ;
    \s__fp \__fp_chk:w #4 #5
  }
```

(*End of definition for* `\__fp_+_o:ww`.)

`\__fp_add_return_ii_o:Nww` Ignore the first operand, and return the second, but using the sign `#1` rather than `#4`. As usual, expand after the floating point.

```
\cs_new:Npn \__fp_add_return_ii_o:Nww #1 #2 ; \s__fp \__fp_chk:w #3 #4
  { \__fp_exp_after_o:w \s__fp \__fp_chk:w #3 #1 }
```

(*End of definition for* `\__fp_add_return_ii_o:Nww`.)

`\__fp_add_zeros_o:Nww` Adding two zeros yields `\c_zero_fp`, except if both zeros were -0.

```
\cs_new:Npn \__fp_add_zeros_o:Nww #1 \s__fp \__fp_chk:w 0 #2
  {
    \if_int_compare:w #2 #1 = 20 \exp_stop_f:
      \exp_after:wN \__fp_add_return_ii_o:Nww
    \else:
      \__fp_case_return_i_o:ww
    \fi:
    #1
    \s__fp \__fp_chk:w 0 #2
  }
```

(*End of definition for* `\__fp_add_zeros_o:Nww`.)

`\__fp_add_inf_o:Nww` If both infinities have the same sign, just return that infinity, otherwise, it is an invalid operation. We find out if that invalid operation is an addition or a subtraction by testing whether the tweaked $\langle sign_2\rangle$ (`#1`) and the $\langle sign_2\rangle$ (`#4`) are identical.

```
\cs_new:Npn \__fp_add_inf_o:Nww
    #1 \s__fp \__fp_chk:w 2 #2 #3; \s__fp \__fp_chk:w 2 #4
  {
    \if_meaning:w #1 #2
      \__fp_case_return_i_o:ww
    \else:
      \__fp_case_use:nw
        {
          \exp_last_unbraced:Nf \__fp_invalid_operation_o:Nww
            { \token_if_eq_meaning:NNTF #1 #4 + - }
        }
    \fi:
    \s__fp \__fp_chk:w 2 #2 #3;
    \s__fp \__fp_chk:w 2 #4
  }
```

(*End of definition for* `\__fp_add_inf_o:Nww`.)

__fp_add_normal_o:Nww

__fp_add_normal_o:Nww ⟨*sign*$_2$⟩ \s__fp __fp_chk:w 1 ⟨*sign*$_1$⟩ ⟨*exp*$_1$⟩ ⟨*body*$_1$⟩ ; \s__fp __fp_chk:w 1 ⟨*initial sign*$_2$⟩ ⟨*exp*$_2$⟩ ⟨*body*$_2$⟩ ;

We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.

```
\cs_new:Npn \__fp_add_normal_o:Nww #1 \s__fp \__fp_chk:w 1 #2
  {
    \if_meaning:w #1#2
      \exp_after:wN \__fp_add_npos_o:NnwNnw
    \else:
      \exp_after:wN \__fp_sub_npos_o:NnwNnw
    \fi:
    #2
  }
```

(*End of definition for* __fp_add_normal_o:Nww.)

74.1.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

__fp_add_npos_o:NnwNnw

__fp_add_npos_o:NnwNnw ⟨*sign*$_1$⟩ ⟨*exp*$_1$⟩ ⟨*body*$_1$⟩ ; \s__fp __fp_chk:w 1 ⟨*initial sign*$_2$⟩ ⟨*exp*$_2$⟩ ⟨*body*$_2$⟩ ;

Since we are doing an addition, the final sign is ⟨*sign*$_1$⟩. Start an __fp_int_eval:w, responsible for computing the exponent: the result, and the ⟨*final sign*⟩ are then given to __fp_sanitize:Nw which checks for overflow. The exponent is computed as the largest exponent #2 or #5, incremented if there is a carry. To add the significands, we decimate the smaller number by the difference between the exponents. This is done by __fp_add_big_i:wNww or __fp_add_big_ii:wNww. We need to bring the final sign with us in the midst of the calculation to round properly at the end.

```
\cs_new:Npn \__fp_add_npos_o:NnwNnw #1#2#3 ; \s__fp \__fp_chk:w 1 #4 #5
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN #1
    \int_value:w \__fp_int_eval:w
      \if_int_compare:w #2 > #5 \exp_stop_f:
        #2
        \exp_after:wN \__fp_add_big_i_o:wNww \int_value:w -
      \else:
        #5
        \exp_after:wN \__fp_add_big_ii_o:wNww \int_value:w
      \fi:
      \__fp_int_eval:w #5 - #2 ; #1 #3;
  }
```

(*End of definition for* __fp_add_npos_o:NnwNnw.)

__fp_add_big_i_o:wNww
__fp_add_big_ii_o:wNww

__fp_add_big_i_o:wNww ⟨*shift*⟩ ; ⟨*final sign*⟩ ⟨*body*$_1$⟩ ; ⟨*body*$_2$⟩ ;

Used in l3fp-expo. Shift the significand of the small number, then add with __fp_add_significand_o:NnnwnnnnN.

```
\cs_new:Npn \__fp_add_big_i_o:wNww #1; #2 #3; #4;
  {
    \__fp_decimate:nNnnnn {#1}
      \__fp_add_significand_o:NnnwnnnnN
```

```
        #4
      #3
      #2
    }
\cs_new:Npn \__fp_add_big_ii_o:wNww #1; #2 #3; #4;
    {
      \__fp_decimate:nNnnnn {#1}
        \__fp_add_significand_o:NnnwnnnnN
        #3
      #4
      #2
    }
```

(*End of definition for* `\__fp_add_big_i_o:wNww` *and* `\__fp_add_big_ii_o:wNww`.)

`\__fp_add_significand_o:NnnwnnnnN`
`\__fp_add_significand_pack:NNNNNNN`
`\__fp_add_significand_test_o:N`

`\__fp_add_significand_o:NnnwnnnnN` ⟨*rounding digit*⟩ {⟨Y'_1⟩} {⟨Y'_2⟩} ⟨*extra-digits*⟩ ; {⟨X_1⟩} {⟨X_2⟩} {⟨X_3⟩} {⟨X_4⟩} ⟨*final sign*⟩

To round properly, we must know at which digit the rounding should occur. This requires to know whether the addition produces an overall carry or not. Thus, we do the computation now and check for a carry, then go back and do the rounding. The rounding may cause a carry in very rare cases such as $0.99\cdots95 \to 1.00\cdots0$, but this situation always give an exact power of 10, for which it is easy to correct the result at the end.

```
\cs_new:Npn \__fp_add_significand_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
    {
      \exp_after:wN \__fp_add_significand_test_o:N
      \int_value:w \__fp_int_eval:w 1#5#6 + #2
        \exp_after:wN \__fp_add_significand_pack:NNNNNNN
        \int_value:w \__fp_int_eval:w 1#7#8 + #3 ; #1
    }
\cs_new:Npn \__fp_add_significand_pack:NNNNNNN #1 #2#3#4#5#6#7
    {
      \if_meaning:w 2 #1
        + 1
      \fi:
      ; #2 #3 #4 #5 #6 #7 ;
    }
\cs_new:Npn \__fp_add_significand_test_o:N #1
    {
      \if_meaning:w 2 #1
        \exp_after:wN \__fp_add_significand_carry_o:wwwNN
      \else:
        \exp_after:wN \__fp_add_significand_no_carry_o:wwwNN
      \fi:
    }
```

(*End of definition for* `\__fp_add_significand_o:NnnwnnnnN`, `\__fp_add_significand_pack:NNNNNNN`, *and* `\__fp_add_significand_test_o:N`.)

`\__fp_add_significand_no_carry_o:wwwNN`

`\__fp_add_significand_no_carry_o:wwwNN` ⟨*8d*⟩ ; ⟨*6d*⟩ ; ⟨*2d*⟩ ; ⟨*rounding digit*⟩ ⟨*sign*⟩

If there's no carry, grab all the digits again and round. The packing function `\__fp_basics_pack_high:NNNNNw` takes care of the case where rounding brings a carry.

```
\cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
      #1; #2; #3#4 ; #5#6
```

```
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \int_value:w \__fp_int_eval:w 1 #1
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \int_value:w \__fp_int_eval:w 1 #2 #3#4
        + \__fp_round:NNN #6 #4 #5
        \exp_after:wN ;
  }
```

(*End of definition for* `\__fp_add_significand_no_carry_o:wwwNN`.)

`\__fp_add_significand_carry_o:wwwNN`

`\__fp_add_significand_carry_o:wwwNN` ⟨*8d*⟩ ; ⟨*6d*⟩ ; ⟨*2d*⟩ ; ⟨*rounding digit*⟩ ⟨*sign*⟩

The case where there is a carry is very similar. Rounding can even raise the first digit from 1 to 2, but we don't care.

```
\cs_new:Npn \__fp_add_significand_carry_o:wwwNN
    #1; #2; #3#4; #5#6
  {
    + 1
    \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw
    \int_value:w \__fp_int_eval:w 1 1 #1
      \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
      \int_value:w \__fp_int_eval:w 1 #2#3 +
        \exp_after:wN \__fp_round:NNN
        \exp_after:wN #6
        \exp_after:wN #3
        \int_value:w \__fp_round_digit:Nw #4 #5 ;
        \exp_after:wN ;
  }
```

(*End of definition for* `\__fp_add_significand_carry_o:wwwNN`.)

74.1.3 Absolute subtraction

`\__fp_sub_npos_o:NnwNnw`
`\__fp_sub_eq_o:Nnwnw`
`\__fp_sub_npos_ii_o:Nnwnw`

`\__fp_sub_npos_o:NnwNnw` ⟨$sign_1$⟩ ⟨exp_1⟩ ⟨$body_1$⟩ ; `\s__fp \__fp_chk:w 1` ⟨*initial* $sign_2$⟩ ⟨exp_2⟩ ⟨$body_2$⟩ ;

Rounding properly in some modes requires to know what the sign of the result will be. Thus, we start by comparing the exponents and significands. If the numbers coincide, return zero. If the second number is larger, swap the numbers and call `\__fp_sub_npos_i_o:Nnwnw` with the opposite of ⟨$sign_1$⟩.

```
\cs_new:Npn \__fp_sub_npos_o:NnwNnw #1#2#3; \s__fp \__fp_chk:w 1 #4#5#6;
  {
    \if_case:w \__fp_compare_npos:nwnw {#2} #3; {#5} #6; \exp_stop_f:
      \exp_after:wN \__fp_sub_eq_o:Nnwnw
    \or:
      \exp_after:wN \__fp_sub_npos_i_o:Nnwnw
    \else:
      \exp_after:wN \__fp_sub_npos_ii_o:Nnwnw
    \fi:
    #1 {#2} #3; {#5} #6;
  }
\cs_new:Npn \__fp_sub_eq_o:Nnwnw #1#2; #3; { \exp_after:wN \c_zero_fp }
\cs_new:Npn \__fp_sub_npos_ii_o:Nnwnw #1 #2; #3;
```

```
  {
    \exp_after:wN \__fp_sub_npos_i_o:Nnwnw
      \int_value:w \__fp_neg_sign:N #1
      #3; #2;
  }
```

(*End of definition for* `\__fp_sub_npos_o:NnwNnw`*,* `\__fp_sub_eq_o:Nnwnw`*, and* `\__fp_sub_npos_ii_o:Nnwnw`*.*)

`\__fp_sub_npos_i_o:Nnwnw`

After the computation is done, `\__fp_sanitize:Nw` checks for overflow/underflow. It expects the ⟨*final sign*⟩ and the ⟨*exponent*⟩ (delimited by `;`). Start an integer expression for the exponent, which starts with the exponent of the largest number, and may be decreased if the two numbers are very close. If the two numbers have the same exponent, call the `near` auxiliary. Otherwise, decimate y, then call the `far` auxiliary to evaluate the difference between the two significands. Note that we decimate by 1 less than one could expect.

```
\cs_new:Npn \__fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN #1
    \int_value:w \__fp_int_eval:w
      #2
      \if_int_compare:w #2 = #4 \exp_stop_f:
        \exp_after:wN \__fp_sub_back_near_o:nnnnnnnnN
      \else:
        \exp_after:wN \__fp_decimate:nNnnnn \exp_after:wN
          { \int_value:w \__fp_int_eval:w #2 - #4 - 1 \exp_after:wN }
          \exp_after:wN \__fp_sub_back_far_o:NnnwnnnnN
      \fi:
        #5
      #3
      #1
  }
```

(*End of definition for* `\__fp_sub_npos_i_o:Nnwnw`*.*)

`\__fp_sub_back_near_o:nnnnnnnnN`
`\__fp_sub_back_near_pack:NNNNNNw`
`\__fp_sub_back_near_after:wNNNNw`

`\__fp_sub_back_near_o:nnnnnnnnN` `{`⟨Y_1⟩`}` `{`⟨Y_2⟩`}` `{`⟨Y_3⟩`}` `{`⟨Y_4⟩`}` `{`⟨X_1⟩`}` `{`⟨X_2⟩`}` `{`⟨X_3⟩`}` `{`⟨X_4⟩`}` ⟨*final sign*⟩

In this case, the subtraction is exact, so we discard the ⟨*final sign*⟩ `#9`. The very large shifts of 10^9 and $1.1 \cdot 10^9$ are unnecessary here, but allow the auxiliaries to be reused later. Each integer expression produces a 10 digit result. If the resulting 16 digits start with a 0, then we need to shift the group, padding with trailing zeros.

```
\cs_new:Npn \__fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
  {
    \exp_after:wN \__fp_sub_back_near_after:wNNNNw
    \int_value:w \__fp_int_eval:w 10#5#6 - #1#2 - 11
      \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
      \int_value:w \__fp_int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
  }
\cs_new:Npn \__fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
  { + #1#2 ; {#3#4#5#6} {#7} ; }
\cs_new:Npn \__fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 ;
  {
    \if_meaning:w 0 #1
```

```
    \exp_after:wN \__fp_sub_back_shift:wnnnn
  \fi:
  ; {#1#2#3#4} {#5}
}
```

(*End of definition for* `\__fp_sub_back_near_o:nnnnnnnnN`*,* `\__fp_sub_back_near_pack:NNNNNNw`*, and* `\__fp_sub_back_near_after:wNNNNw`*.*)

`\__fp_sub_back_shift:wnnnn`
`\__fp_sub_back_shift_ii:ww`
`\__fp_sub_back_shift_iii:NNNNNNNNw`
`\__fp_sub_back_shift_iv:nnnnw`

`\__fp_sub_back_shift:wnnnn ;` {⟨Z_1⟩} {⟨Z_2⟩} {⟨Z_3⟩} {⟨Z_4⟩} `;`

This function is called with ⟨Z_1⟩ ≤ 999. Act with `\number` to trim leading zeros from ⟨Z_1⟩ ⟨Z_2⟩ (we don't do all four blocks at once, since non-zero blocks would then overflow TeX's integers). If the first two blocks are zero, the auxiliary receives an empty `#1` and trims `#2#30` from leading zeros, yielding a total shift between 7 and 16 to the exponent. Otherwise we get the shift from `#1` alone, yielding a result between 1 and 6. Once the exponent is taken care of, trim leading zeros from `#1#2#3` (when `#1` is empty, the space before `#2#3` is ignored), get four blocks of 4 digits and finally clean up. Trailing zeros are added so that digits can be grabbed safely.

```
\cs_new:Npn \__fp_sub_back_shift:wnnnn ; #1#2
  {
    \exp_after:wN \__fp_sub_back_shift_ii:ww
    \int_value:w #1 #2 0 ;
  }
\cs_new:Npn \__fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
  {
    \if_meaning:w @ #1 @
      - 7
      - \exp_after:wN \use_i:nnn
        \exp_after:wN \__fp_sub_back_shift_iii:NNNNNNNNw
        \int_value:w #2#3 0 ~ 123456789;
    \else:
      - \__fp_sub_back_shift_iii:NNNNNNNNw #1 123456789;
    \fi:
    \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
    \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
    \exp_after:wN \__fp_sub_back_shift_iv:nnnnw
    \exp_after:wN ;
    \int_value:w
    #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
  }
\cs_new:Npn \__fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
\cs_new:Npn \__fp_sub_back_shift_iv:nnnnw #1 ; #2 ; { ; #1 ; }
```

(*End of definition for* `\__fp_sub_back_shift:wnnnn` *and others.*)

`\__fp_sub_back_far_o:NnnwnnnnN`

`\__fp_sub_back_far_o:NnnwnnnnN` ⟨*rounding*⟩ {⟨Y'_1⟩} {⟨Y'_2⟩} ⟨*extra-digits*⟩ `;` {⟨X_1⟩} {⟨X_2⟩} {⟨X_3⟩} {⟨X_4⟩} ⟨*final sign*⟩

If the difference is greater than $10^{\langle expo_x \rangle}$, call the `very_far` auxiliary. If the result is less than $10^{\langle expo_x \rangle}$, call the `not_far` auxiliary. If it is too close a call to know yet, namely if $1\langle Y'_1\rangle\langle Y'_2\rangle = \langle X_1\rangle\langle X_2\rangle\langle X_3\rangle\langle X_4\rangle 0$, then call the `quite_far` auxiliary. We use the odd combination of space and semi-colon delimiters to allow the `not_far` auxiliary to grab each piece individually, the `very_far` auxiliary to use `\__fp_pack_eight:wNNNNNNNN`, and the `quite_far` to ignore the significands easily (using the `;` delimiter).

```
\cs_new:Npn \__fp_sub_back_far_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
```

```
  {
    \if_case:w
      \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
        \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
          0
        \else:
          \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: 1
        \fi:
      \else:
        \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: 1
      \fi:
      \exp_stop_f:
           \exp_after:wN \__fp_sub_back_quite_far_o:wwNN
    \or:   \exp_after:wN \__fp_sub_back_very_far_o:wwwwNN
    \else: \exp_after:wN \__fp_sub_back_not_far_o:wwwwNN
    \fi:
    #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1
  }
```

(*End of definition for* `\__fp_sub_back_far_o:NnnwnnnnN`.)

`\__fp_sub_back_quite_far_o:wwNN`
`\__fp_sub_back_quite_far_ii:NN`

The easiest case is when $x-y$ is extremely close to a power of 10, namely the first digit of x is 1, and all others vanish when subtracting y. Then the ⟨*rounding*⟩ `#3` and the ⟨*final sign*⟩ `#4` control whether we get 1 or 0.9999999999999999. In the usual round-to-nearest mode, we get 1 whenever the ⟨*rounding*⟩ digit is less than or equal to 5 (remember that the ⟨*rounding*⟩ digit is only equal to 5 if there was no further non-zero digit).

```
\cs_new:Npn \__fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
  {
    \exp_after:wN \__fp_sub_back_quite_far_ii:NN
    \exp_after:wN #3
    \exp_after:wN #4
  }
\cs_new:Npn \__fp_sub_back_quite_far_ii:NN #1#2
  {
    \if_case:w \__fp_round_neg:NNN #2 0 #1
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
      { ; {1000} {0000} {0000} {0000} ; }
      { - 1 ; {9999} {9999} {9999} {9999} ; }
  }
```

(*End of definition for* `\__fp_sub_back_quite_far_o:wwNN` *and* `\__fp_sub_back_quite_far_ii:NN`.)

`\__fp_sub_back_not_far_o:wwwwNN`

In the present case, x and y have different exponents, but y is large enough that $x-y$ has a smaller exponent than x. Decrement the exponent (with `-1`). Then proceed in a way similar to the `near` auxiliaries seen earlier, but multiplying x by 10 (`#30` and `#40` below), and with the added quirk that the ⟨*rounding*⟩ digit has to be taken into account. Namely, we may have to decrease the result by one unit if `\__fp_round_neg:NNN` returns 1. This function expects the ⟨*final sign*⟩ `#6`, the last digit of `1100000000+#40-#2`, and the ⟨*rounding*⟩ digit. Instead of redoing the computation for the second argument, we note that `\__fp_round_neg:NNN` only cares about its parity, which is identical to that of the last digit of `#2`.

```
\cs_new:Npn \__fp_sub_back_not_far_o:wwwwNN #1 ~ #2; #3 ~ #4; #5#6
  {
    - 1
    \exp_after:wN \__fp_sub_back_near_after:wNNNNw
    \int_value:w \__fp_int_eval:w 1#30 - #1 - 11
      \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
      \int_value:w \__fp_int_eval:w 11 0000 0000 + #40 - #2
        - \exp_after:wN \__fp_round_neg:NNN
          \exp_after:wN #6
          \use_none:nnnnnnn #2 #5
        \exp_after:wN ;
  }
```

(*End of definition for* `\__fp_sub_back_not_far_o:wwwwNN`*.*)

`\__fp_sub_back_very_far_o:wwwwNN`
`\__fp_sub_back_very_far_ii_o:nnNwwNN`

The case where $x - y$ and x have the same exponent is a bit more tricky, mostly because it cannot reuse the same auxiliaries. Shift the y significand by adding a leading 0. Then the logic is similar to the `not_far` functions above. Rounding is a bit more complicated: we have two ⟨*rounding*⟩ digits `#3` and `#6` (from the decimation, and from the new shift) to take into account, and getting the parity of the main result requires a computation. The first `\int_value:w` triggers the second one because the number is unfinished; we can thus not use 0 in place of 2 there.

```
\cs_new:Npn \__fp_sub_back_very_far_o:wwwwNN #1#2#3#4#5#6#7
  {
    \__fp_pack_eight:wNNNNNNNN
    \__fp_sub_back_very_far_ii_o:nnNwwNN
    { 0 #1#2#3 #4#5#6#7 }
    ;
  }
\cs_new:Npn \__fp_sub_back_very_far_ii_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \int_value:w \__fp_int_eval:w 1#4 - #1 - 1
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \int_value:w \__fp_int_eval:w 2#5 - #2
        - \exp_after:wN \__fp_round_neg:NNN
          \exp_after:wN #7
          \int_value:w
            \if_int_odd:w \__fp_int_eval:w #5 - #2 \__fp_int_eval_end:
              1 \else: 2 \fi:
          \int_value:w \__fp_round_digit:Nw #3 #6 ;
      \exp_after:wN ;
  }
```

(*End of definition for* `\__fp_sub_back_very_far_o:wwwwNN` *and* `\__fp_sub_back_very_far_ii_o:nnNwwNN`*.*)

74.2 Multiplication

74.2.1 Signs, and special numbers

`\__fp_*_o:ww`

We go through an auxiliary, which is common with `\__fp_/_o:ww`. The first argument is the operation, used for the invalid operation exception. The second is inserted in a formula to dispatch cases slightly differently between multiplication and division. The third is the operation for normal floating points. The fourth is there for extra cases needed in `\__fp_/_o:ww`.

```
\cs_new:cpn { __fp_*_o:ww }
  {
    \__fp_mul_cases_o:NnNnww
      *
      { - 2 + }
      \__fp_mul_npos_o:Nww
      { }
  }
```

(*End of definition for* `\__fp_*_o:ww`.)

`\__fp_mul_cases_o:nNnnww` Split into 10 cases (12 for division). If both numbers are normal, go to case 0 (same sign) or case 1 (opposite signs): in both cases, call `\__fp_mul_npos_o:Nww` to do the work. If the first operand is `nan`, go to case 2, in which the second operand is discarded; if the second operand is `nan`, go to case 3, in which the first operand is discarded (note the weird interaction with the final test on signs). Then we separate the case where the first number is normal and the second is zero: this goes to cases 4 and 5 for multiplication, 10 and 11 for division. Otherwise, we do a computation which dispatches the products $0\times 0 = 0\times 1 = 1\times 0 = 0$ to case 4 or 5 depending on the combined sign, the products $0\times\infty$ and $\infty\times 0$ to case 6 or 7 (invalid operation), and the products $1\times\infty = \infty\times 1 = \infty\times\infty = \infty$ to cases 8 and 9. Note that the code for these two cases (which return $\pm\infty$) is inserted as argument `#4`, because it differs in the case of divisions.

```
\cs_new:Npn \__fp_mul_cases_o:NnNnww
    #1#2#3#4 \s__fp \__fp_chk:w #5#6#7; \s__fp \__fp_chk:w #8#9
  {
    \if_case:w \__fp_int_eval:w
                 \if_int_compare:w #5 #8 = 11 ~
                   1
                 \else:
                   \if_meaning:w 3 #8
                     3
                   \else:
                     \if_meaning:w 3 #5
                       2
                     \else:
                       \if_int_compare:w #5 #8 = 10 ~
                         9 #2 - 2
                       \else:
                         (#5 #2 #8) / 2 * 2 + 7
                       \fi:
                     \fi:
                   \fi:
                 \fi:
                 \if_meaning:w #6 #9 - 1 \fi:
               \__fp_int_eval_end:
         \__fp_case_use:nw { #3 0 }
    \or: \__fp_case_use:nw { #3 2 }
    \or: \__fp_case_return_i_o:ww
    \or: \__fp_case_return_ii_o:ww
    \or: \__fp_case_return_o:Nww \c_zero_fp
    \or: \__fp_case_return_o:Nww \c_minus_zero_fp
```

```
    \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
    \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
    \or: \__fp_case_return_o:Nww \c_inf_fp
    \or: \__fp_case_return_o:Nww \c_minus_inf_fp
    #4
    \fi:
    \s__fp \__fp_chk:w #5 #6 #7;
    \s__fp \__fp_chk:w #8 #9
  }
```

(*End of definition for* `\__fp_mul_cases_o:nNnnww`.)

74.2.2 Absolute multiplication

In this subsection, we perform the multiplication of two positive normal numbers.

`\__fp_mul_npos_o:Nww`

`\__fp_mul_npos_o:Nww` ⟨*final sign*⟩ `\s__fp` `\__fp_chk:w` 1 ⟨*sign*$_1$⟩ {⟨*exp*$_1$⟩} ⟨*body*$_1$⟩ ; `\s__fp` `\__fp_chk:w` 1 ⟨*sign*$_2$⟩ {⟨*exp*$_2$⟩} ⟨*body*$_2$⟩ ;

After the computation, `\__fp_sanitize:Nw` checks for overflow or underflow. As we did for addition, `\__fp_int_eval:w` computes the exponent, catching any shift coming from the computation in the significand. The ⟨*final sign*⟩ is needed to do the rounding properly in the significand computation. We setup the post-expansion here, triggered by `\__fp_mul_significand_o:nnnnNnnnn`.

This is also used in l3fp-convert.

```
\cs_new:Npn \__fp_mul_npos_o:Nww
    #1 \s__fp \__fp_chk:w #2 #3 #4 #5 ; \s__fp \__fp_chk:w #6 #7 #8 #9 ;
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN #1
    \int_value:w \__fp_int_eval:w
      #4 + #8
      \__fp_mul_significand_o:nnnnNnnnn #5 #1 #9
  }
```

(*End of definition for* `\__fp_mul_npos_o:Nww`.)

`\__fp_mul_significand_o:nnnnNnnnn`
`\__fp_mul_significand_drop:NNNNNw`
`\__fp_mul_significand_keep:NNNNNw`

`\__fp_mul_significand_o:nnnnNnnnn` {⟨X_1⟩} {⟨X_2⟩} {⟨X_3⟩} {⟨X_4⟩} ⟨*sign*⟩ {⟨Y_1⟩} {⟨Y_2⟩} {⟨Y_3⟩} {⟨Y_4⟩}

Note the three semicolons at the end of the definition. One is for the last `\__fp_mul_significand_drop:NNNNNw`; one is for `\__fp_round_digit:Nw` later on; and one, preceded by `\exp_after:wN`, which is correctly expanded (within an `\__fp_int_eval:w`), is used by `\__fp_basics_pack_low:NNNNNw`.

The product of two 16 digit integers has 31 or 32 digits, but it is impossible to know which one before computing. The place where we round depends on that number of digits, and may depend on all digits until the last in some rare cases. The approach is thus to compute the 5 first blocks of 4 digits (the first one is between 100 and 9999 inclusive), and a compact version of the remaining 3 blocks. Afterwards, the number of digits is known, and we can do the rounding within yet another set of `\__fp_int_eval:w`.

```
\cs_new:Npn \__fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
  {
    \exp_after:wN \__fp_mul_significand_test_f:NNN
    \exp_after:wN #5
    \int_value:w \__fp_int_eval:w 99990000 + #1*#6 +
```

```
      \exp_after:wN \__fp_mul_significand_keep:NNNNNw
      \int_value:w \__fp_int_eval:w 99990000 + #1*#7 + #2*#6 +
        \exp_after:wN \__fp_mul_significand_keep:NNNNNw
        \int_value:w \__fp_int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
          \exp_after:wN \__fp_mul_significand_drop:NNNNNw
          \int_value:w \__fp_int_eval:w 99990000 + #1*#9 + #2*#8 +
            #3*#7 + #4*#6 +
            \exp_after:wN \__fp_mul_significand_drop:NNNNNw
            \int_value:w \__fp_int_eval:w 99990000 + #2*#9 + #3*#8 +
              #4*#7 +
              \exp_after:wN \__fp_mul_significand_drop:NNNNNw
              \int_value:w \__fp_int_eval:w 99990000 + #3*#9 + #4*#8 +
                \exp_after:wN \__fp_mul_significand_drop:NNNNNw
                \int_value:w \__fp_int_eval:w 100000000 + #4*#9 ;
    ; \exp_after:wN ;
  }
\cs_new:Npn \__fp_mul_significand_drop:NNNNNw #1#2#3#4#5 #6;
  { #1#2#3#4#5 ; + #6 }
\cs_new:Npn \__fp_mul_significand_keep:NNNNNw #1#2#3#4#5 #6;
  { #1#2#3#4#5 ; #6 ; }
```

(*End of definition for* `\__fp_mul_significand_o:nnnnNnnnn`*,* `\__fp_mul_significand_drop:NNNNNw`*, and* `\__fp_mul_significand_keep:NNNNNw`*.*)

__fp_mul_significand_test_f:NNN

`\__fp_mul_significand_test_f:NNN` ⟨*sign*⟩ `1` ⟨*digits 1-8*⟩ `;` ⟨*digits 9-12*⟩ `;` ⟨*digits 13-16*⟩ `; +` ⟨*digits 17-20*⟩ `+` ⟨*digits 21-24*⟩ `+` ⟨*digits 25-28*⟩ `+` ⟨*digits 29-32*⟩ `; \exp_after:wN ;`

If the ⟨*digit 1*⟩ is non-zero, then for rounding we only care about the digits 16 and 17, and whether further digits are zero or not (check for exact ties). On the other hand, if ⟨*digit 1*⟩ is zero, we care about digits 17 and 18, and whether further digits are zero.

```
\cs_new:Npn \__fp_mul_significand_test_f:NNN #1 #2 #3
  {
    \if_meaning:w 0 #3
      \exp_after:wN \__fp_mul_significand_small_f:NNwwwN
    \else:
      \exp_after:wN \__fp_mul_significand_large_f:NwwNNNN
    \fi:
    #1 #3
  }
```

(*End of definition for* `\__fp_mul_significand_test_f:NNN`*.*)

__fp_mul_significand_large_f:NwwNNNN

In this branch, ⟨*digit 1*⟩ is non-zero. The result is thus ⟨*digits 1-16*⟩, plus some rounding which depends on the digits 16, 17, and whether all subsequent digits are zero or not. Here, `\__fp_round_digit:Nw` takes digits 17 and further (as an integer expression), and replaces it by a ⟨*rounding digit*⟩, suitable for `\__fp_round:NNN`.

```
\cs_new:Npn \__fp_mul_significand_large_f:NwwNNNN #1 #2; #3; #4#5#6#7; +
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \int_value:w \__fp_int_eval:w 1#2
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \int_value:w \__fp_int_eval:w 1#3#4#5#6#7
        + \exp_after:wN \__fp_round:NNN
          \exp_after:wN #1
```

```
          \exp_after:wN #7
          \int_value:w \__fp_round_digit:Nw
  }
```

(*End of definition for* `\__fp_mul_significand_large_f:NwwNNNN`.)

`\__fp_mul_significand_small_f:NNwwwN` In this branch, ⟨*digit 1*⟩ is zero. Our result is thus ⟨*digits 2-17*⟩, plus some rounding which depends on the digits 17, 18, and whether all subsequent digits are zero or not. The 8 digits `1#3` are followed, after expansion of the `small_pack` auxiliary, by the next digit, to form a 9 digit number.

```
\cs_new:Npn \__fp_mul_significand_small_f:NNwwwN #1 #2#3; #4#5; #6; + #7
  {
    - 1
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \int_value:w \__fp_int_eval:w 1#3#4
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \int_value:w \__fp_int_eval:w 1#5#6#7
        + \exp_after:wN \__fp_round:NNN
          \exp_after:wN #1
          \exp_after:wN #7
          \int_value:w \__fp_round_digit:Nw
  }
```

(*End of definition for* `\__fp_mul_significand_small_f:NNwwwN`.)

74.3 Division

74.3.1 Signs, and special numbers

Time is now ripe to tackle the hardest of the four elementary operations: division.

`\__fp_/_o:ww` Filtering special floating point is very similar to what we did for multiplications, with a few variations. Invalid operation exceptions display `/` rather than `*`. In the formula for dispatch, we replace `- 2 +` by `-`. The case of normal numbers is treated using `\__fp_div_npos_o:Nww` rather than `\__fp_mul_npos_o:Nww`. There are two additional cases: if the first operand is normal and the second is a zero, then the division by zero exception is raised: cases 10 and 11 of the `\if_case:w` construction in `\__fp_mul_cases_o:NnNnww` are provided as the fourth argument here.

```
\cs_new:cpn { __fp_/_o:ww }
  {
    \__fp_mul_cases_o:NnNnww
      /
      { - }
      \__fp_div_npos_o:Nww
      {
        \or:
          \__fp_case_use:nw
            { \__fp_division_by_zero_o:NNww \c_inf_fp / }
        \or:
          \__fp_case_use:nw
            { \__fp_division_by_zero_o:NNww \c_minus_inf_fp / }
      }
  }
```

(*End of definition for* `\__fp_/_o:ww`.)

`\__fp_div_npos_o:Nww`

`\__fp_div_npos_o:Nww` ⟨*final sign*⟩ `\s__fp` `\__fp_chk:w` 1 ⟨$sign_A$⟩ {⟨*exp A*⟩} {⟨A_1⟩} {⟨A_2⟩} {⟨A_3⟩} {⟨A_4⟩} ; `\s__fp` `\__fp_chk:w` 1 ⟨$sign_Z$⟩ {⟨*exp Z*⟩} {⟨Z_1⟩} {⟨Z_2⟩} {⟨Z_3⟩} {⟨Z_4⟩} ;

We want to compute A/Z. As for multiplication, `\__fp_sanitize:Nw` checks for overflow or underflow; we provide it with the ⟨*final sign*⟩, and an integer expression in which we compute the exponent. We set up the arguments of `\__fp_div_significand_i_o:wnnw`, namely an integer ⟨y⟩ obtained by adding 1 to the first 5 digits of Z (explanation given soon below), then the four {⟨A_i⟩}, then the four {⟨Z_i⟩}, a semi-colon, and the ⟨*final sign*⟩, used for rounding at the end.

```
\cs_new:Npn \__fp_div_npos_o:Nww
    #1 \s__fp \__fp_chk:w 1 #2 #3 #4 ; \s__fp \__fp_chk:w 1 #5 #6 #7#8#9;
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN #1
    \int_value:w \__fp_int_eval:w
      #3 - #6
      \exp_after:wN \__fp_div_significand_i_o:wnnw
        \int_value:w \__fp_int_eval:w #7 \use_i:nnnn #8 + 1 ;
        #4
        {#7}{#8}#9 ;
        #1
  }
```

(*End of definition for* `\__fp_div_npos_o:Nww`.)

74.3.2 Work plan

In this subsection, we explain how to avoid overflowing TeX's integers when performing the division of two positive normal numbers.

We are given two numbers, $A = 0.A_1A_2A_3A_4$ and $Z = 0.Z_1Z_2Z_3Z_4$, in blocks of 4 digits, and we know that the first digits of A_1 and of Z_1 are non-zero. To compute A/Z, we proceed as follows.

- Find an integer $Q_A \simeq 10^4A/Z$.
- Replace A by $B = 10^4A - Q_AZ$.
- Find an integer $Q_B \simeq 10^4B/Z$.
- Replace B by $C = 10^4B - Q_BZ$.
- Find an integer $Q_C \simeq 10^4C/Z$.
- Replace C by $D = 10^4C - Q_CZ$.
- Find an integer $Q_D \simeq 10^4D/Z$.
- Consider $E = 10^4D - Q_DZ$, and ensure correct rounding.

The result is then $Q = 10^{-4}Q_A + 10^{-8}Q_B + 10^{-12}Q_C + 10^{-16}Q_D + \text{rounding}$. Since the Q_i are integers, B, C, D, and E are all exact multiples of 10^{-16}, in other words, computing with 16 digits after the decimal separator yields exact results. The problem is the risk of overflow: in general B, C, D, and E may be greater than 1.

Unfortunately, things are not as easy as they seem. In particular, we want all intermediate steps to be positive, since negative results would require extra calculations at the end. This requires that $Q_A \leq 10^4 A/Z$ *etc.* A reasonable attempt would be to define Q_A as

$$\texttt{\textbackslash int\_eval:n}\left\{\frac{A_1A_2}{Z_1+1} - 1\right\} \leq 10^4 \frac{A}{Z}$$

Subtracting 1 at the end takes care of the fact that ε-TEX's `\__fp_int_eval:w` rounds divisions instead of truncating (really, 1/2 would be sufficient, but we work with integers). We add 1 to Z_1 because $Z_1 \leq 10^4 Z < Z_1 + 1$ and we need Q_A to be an underestimate. However, we are now underestimating Q_A too much: it can be wrong by up to 100, for instance when $Z = 0.1$ and $A \simeq 1$. Then B could take values up to 10 (maybe more), and a few steps down the line, we would run into arithmetic overflow, since TEX can only handle integers less than roughly $2 \cdot 10^9$.

A better formula is to take

$$Q_A = \texttt{\textbackslash int\_eval:n}\left\{\frac{10 \cdot A_1A_2}{\lfloor 10^{-3} \cdot Z_1Z_2 \rfloor + 1} - 1\right\}.$$

This is always less than $10^9 A/(10^5 Z)$, as we wanted. In words, we take the 5 first digits of Z into account, and the 8 first digits of A, using 0 as a 9-th digit rather than the true digit for efficiency reasons. We shall prove that using this formula to define all the Q_i avoids any overflow. For convenience, let us denote

$$y = \left\lfloor 10^{-3} \cdot Z_1Z_2 \right\rfloor + 1,$$

so that, taking into account the fact that ε-TEX rounds ties away from zero,

$$\begin{aligned} Q_A &= \left\lfloor \frac{A_1A_20}{y} - \frac{1}{2} \right\rfloor \\ &> \frac{A_1A_20}{y} - \frac{3}{2}. \end{aligned}$$

Note that $10^4 < y \leq 10^5$, and $999 \leq Q_A \leq 99989$. Also note that this formula does not cause an overflow as long as $A < (2^{31} - 1)/10^9 \simeq 2.147\cdots$, since the numerator involves an integer slightly smaller than $10^9 A$.

Let us bound B:

$$\begin{aligned} 10^5 B &= A_1A_20 + 10 \cdot 0.A_3A_4 - 10 \cdot Z_1.Z_2Z_3Z_4 \cdot Q_A \\ &< A_1A_20 \cdot \left(1 - 10 \cdot \frac{Z_1.Z_2Z_3Z_4}{y}\right) + \frac{3}{2} \cdot 10 \cdot Z_1.Z_2Z_3Z_4 + 10 \\ &\leq \frac{A_1A_20 \cdot (y - 10 \cdot Z_1.Z_2Z_3Z_4)}{y} + \frac{3}{2}y + 10 \\ &\leq \frac{A_1A_20 \cdot 1}{y} + \frac{3}{2}y + 10 \leq \frac{10^9 A}{y} + 1.6 \cdot y. \end{aligned}$$

At the last step, we hide 10 into the second term for later convenience. The same reasoning yields

$$\begin{aligned}
10^5 B &< 10^9 A/y + 1.6y,\\
10^5 C &< 10^9 B/y + 1.6y,\\
10^5 D &< 10^9 C/y + 1.6y,\\
10^5 E &< 10^9 D/y + 1.6y.
\end{aligned}$$

The goal is now to prove that none of B, C, D, and E can go beyond $(2^{31} - 1)/10^9 = 2.147\cdots$.

Combining the various inequalities together with $A < 1$, we get

$$\begin{aligned}
10^5 B &< 10^9/y + 1.6y,\\
10^5 C &< 10^{13}/y^2 + 1.6(y + 10^4),\\
10^5 D &< 10^{17}/y^3 + 1.6(y + 10^4 + 10^8/y),\\
10^5 E &< 10^{21}/y^4 + 1.6(y + 10^4 + 10^8/y + 10^{12}/y^2).
\end{aligned}$$

All of those bounds are convex functions of y (since every power of y involved is convex, and the coefficients are positive), and thus maximal at one of the end-points of the allowed range $10^4 < y \leq 10^5$. Thus,

$$\begin{aligned}
10^5 B &< \max(1.16 \cdot 10^5, 1.7 \cdot 10^5),\\
10^5 C &< \max(1.32 \cdot 10^5, 1.77 \cdot 10^5),\\
10^5 D &< \max(1.48 \cdot 10^5, 1.777 \cdot 10^5),\\
10^5 E &< \max(1.64 \cdot 10^5, 1.7777 \cdot 10^5).
\end{aligned}$$

All of those bounds are less than $2.147 \cdot 10^5$, and we are thus within TeX's bounds in all cases!

We later need to have a bound on the Q_i. Their definitions imply that $Q_A < 10^9 A/y - 1/2 < 10^5 A$ and similarly for the other Q_i. Thus, all of them are less than 177770.

The last step is to ensure correct rounding. We have

$$A/Z = \sum_{i=1}^{4} \left(10^{-4i} Q_i\right) + 10^{-16} E/Z$$

exactly. Furthermore, we know that the result is in $[0.1, 10)$, hence will be rounded to a multiple of 10^{-16} or of 10^{-15}, so we only need to know the integer part of E/Z, and a "rounding" digit encoding the rest. Equivalently, we need to find the integer part of $2E/Z$, and determine whether it was an exact integer or not (this serves to detect ties). Since

$$\frac{2E}{Z} = 2\frac{10^5 E}{10^5 Z} \leq 2\frac{10^5 E}{10^4} < 36,$$

this integer part is between 0 and 35 inclusive. We let ε-TeX round

$$P = \texttt{\int_eval:n} \left\{ \frac{2 \cdot E_1 E_2}{Z_1 Z_2} \right\},$$

which differs from $2E/Z$ by at most

$$\frac{1}{2} + 2 \left| \frac{E}{Z} - \frac{E}{10^{-8} Z_1 Z_2} \right| + 2 \left| \frac{10^8 E - E_1 E_2}{Z_1 Z_2} \right| < 1,$$

($1/2$ comes from ε-TeX's rounding) because each absolute value is less than 10^{-7}. Thus P is either the correct integer part, or is off by 1; furthermore, if $2E/Z$ is an integer, $P = 2E/Z$. We will check the sign of $2E - PZ$. If it is negative, then $E/Z \in \big((P-1)/2, P/2\big)$. If it is zero, then $E/Z = P/2$. If it is positive, then $E/Z \in \big(P/2, (P-1)/2\big)$. In each case, we know how to round to an integer, depending on the parity of P, and the rounding mode.

74.3.3 Implementing the significand division

__fp_div_significand_i_o:wnnw

`\__fp_div_significand_i_o:wnnw` ⟨*y*⟩ `;` `{`⟨A_1⟩`}` `{`⟨A_2⟩`}` `{`⟨A_3⟩`}` `{`⟨A_4⟩`}` `{`⟨Z_1⟩`}` `{`⟨Z_2⟩`}` `{`⟨Z_3⟩`}` `{`⟨Z_4⟩`}` `;` ⟨*sign*⟩

Compute $10^6 + Q_A$ (a 7 digit number thanks to the shift), unbrace ⟨A_1⟩ and ⟨A_2⟩, and prepare the ⟨*continuation*⟩ arguments for 4 consecutive calls to `\__fp_div_significand_calc:wwnnnnnnn`. Each of these calls needs ⟨*y*⟩ (`#1`), and it turns out that we need post-expansion there, hence the `\int_value:w`. Here, `#4` is six brace groups, which give the six first `n`-type arguments of the `calc` function.

```
\cs_new:Npn \__fp_div_significand_i_o:wnnw #1 ; #2#3 #4 ;
  {
    \exp_after:wN \__fp_div_significand_test_o:w
    \int_value:w \__fp_int_eval:w
      \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
      \int_value:w \__fp_int_eval:w 999999 + #2 #3 0 / #1 ;
        #2 #3 ;
        #4
        { \exp_after:wN \__fp_div_significand_ii:wwn \int_value:w #1 }
        { \exp_after:wN \__fp_div_significand_ii:wwn \int_value:w #1 }
        { \exp_after:wN \__fp_div_significand_ii:wwn \int_value:w #1 }
        { \exp_after:wN \__fp_div_significand_iii:wwnnnnn \int_value:w #1 }
  }
```

(*End of definition for* `\__fp_div_significand_i_o:wnnw`*.*)

__fp_div_significand_calc:wwnnnnnnn
__fp_div_significand_calc_i:wwnnnnnnn
__fp_div_significand_calc_ii:wwnnnnnnn

`\__fp_div_significand_calc:wwnnnnnnn` ⟨$10^6 + Q_A$⟩ `;` ⟨A_1⟩ ⟨A_2⟩ `;` `{`⟨A_3⟩`}` `{`⟨A_4⟩`}` `{`⟨Z_1⟩`}` `{`⟨Z_2⟩`}` `{`⟨Z_3⟩`}` `{`⟨Z_4⟩`}` `{`⟨*continuation*⟩`}`

expands to

⟨$10^6 + Q_A$⟩ ⟨*continuation*⟩ `;` ⟨B_1⟩ ⟨B_2⟩ `;` `{`⟨B_3⟩`}` `{`⟨B_4⟩`}` `{`⟨Z_1⟩`}` `{`⟨Z_2⟩`}` `{`⟨Z_3⟩`}` `{`⟨Z_4⟩`}`

where $B = 10^4 A - Q_A \cdot Z$. This function is also used to compute C, D, E (with the input shifted accordingly), and is used in l3fp-expo.

We know that $0 < Q_A < 1.8 \cdot 10^5$, so the product of Q_A with each Z_i is within TeX's bounds. However, it is a little bit too large for our purposes: we would not be able to use the usual trick of adding a large power of 10 to ensure that the number of digits is fixed.

The bound on Q_A, implies that $10^6 + Q_A$ starts with the digit 1, followed by 0 or 1. We test, and call different auxiliaries for the two cases. An earlier implementation did the tests within the computation, but since we added a $\langle$*continuation*$\rangle$, this is not possible because the macro has 9 parameters.

The result we want is then (the overall power of 10 is arbitrary):

$$10^{-4}(\#2 - \#1 \cdot \#5 - 10 \cdot \langle i \rangle \cdot \#5\#6) + 10^{-8}(\#3 - \#1 \cdot \#6 - 10 \cdot \langle i \rangle \cdot \#7)$$
$$+ 10^{-12}(\#4 - \#1 \cdot \#7 - 10 \cdot \langle i \rangle \cdot \#8) + 10^{-16}(-\#1 \cdot \#8),$$

where $\langle i \rangle$ stands for the 10^5 digit of Q_A, which is 0 or 1, and #1, #2, *etc.* are the parameters of either auxiliary. The factors of 10 come from the fact that $Q_A = 10 \cdot 10^4 \cdot \langle i \rangle + \#1$. As usual, to combine all the terms, we need to choose some shifts which must ensure that the number of digits of the second, third, and fourth terms are each fixed. Here, the positive contributions are at most 10^8 and the negative contributions can go up to 10^9. Indeed, for the auxiliary with $\langle i \rangle = 1$, `#1` is at most 80000, leading to contributions of at worse $-8 \cdot 10^8 4$, while the other negative term is very small $< 10^6$ (except in the first expression, where we don't care about the number of digits); for the auxiliary with $\langle i \rangle = 0$, `#1` can go up to 99999, but there is no other negative term. Hence, a good choice is $2 \cdot 10^9$, which produces totals in the range $[10^9, 2.1 \cdot 10^9]$. We are flirting with TEX's limits once more.

```
\cs_new:Npn \__fp_div_significand_calc:wwnnnnnnn 1#1
  {
    \if_meaning:w 1 #1
      \exp_after:wN \__fp_div_significand_calc_i:wwnnnnnnn
    \else:
      \exp_after:wN \__fp_div_significand_calc_ii:wwnnnnnnn
    \fi:
  }
\cs_new:Npn \__fp_div_significand_calc_i:wwnnnnnnn
  #1; #2;#3#4 #5#6#7#8 #9
  {
    1 1 #1
    #9 \exp_after:wN ;
    \int_value:w \__fp_int_eval:w \c__fp_Bigg_leading_shift_int
      + #2 - #1 * #5 - #5#60
      \exp_after:wN \__fp_pack_Bigg:NNNNNNw
      \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
        + #3 - #1 * #6 - #70
        \exp_after:wN \__fp_pack_Bigg:NNNNNNw
        \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
          + #4 - #1 * #7 - #80
          \exp_after:wN \__fp_pack_Bigg:NNNNNNw
          \int_value:w \__fp_int_eval:w \c__fp_Bigg_trailing_shift_int
            - #1 * #8 ;
    {#5}{#6}{#7}{#8}
  }
\cs_new:Npn \__fp_div_significand_calc_ii:wwnnnnnnn
  #1; #2;#3#4 #5#6#7#8 #9
  {
    1 0 #1
```

```
    #9 \exp_after:wN ;
    \int_value:w \__fp_int_eval:w \c__fp_Bigg_leading_shift_int
      + #2 - #1 * #5
      \exp_after:wN \__fp_pack_Bigg:NNNNNNw
      \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
        + #3 - #1 * #6
        \exp_after:wN \__fp_pack_Bigg:NNNNNNw
        \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
          + #4 - #1 * #7
          \exp_after:wN \__fp_pack_Bigg:NNNNNNw
          \int_value:w \__fp_int_eval:w \c__fp_Bigg_trailing_shift_int
            - #1 * #8 ;
    {#5}{#6}{#7}{#8}
  }
```

(*End of definition for* `\__fp_div_significand_calc:wwnnnnnnn`*,* `\__fp_div_significand_calc_i:wwnnnnnnn`*, and* `\__fp_div_significand_calc_ii:wwnnnnnnn`*.*)

`\__fp_div_significand_ii:wwn`

`\__fp_div_significand_ii:wwn` ⟨y⟩ `;` ⟨B_1⟩ `;` `{`⟨B_2⟩`}` `{`⟨B_3⟩`}` `{`⟨B_4⟩`}` `{`⟨Z_1⟩`}` `{`⟨Z_2⟩`}` `{`⟨Z_3⟩`}` `{`⟨Z_4⟩`}` ⟨*continuations*⟩ ⟨*sign*⟩

Compute Q_B by evaluating ⟨B_1⟩⟨B_2⟩$0/y-1$. The result is output to the left, in an `\__fp_int_eval:w` which we start now. Once that is evaluated (and the other Q_i also, since later expansions are triggered by this one), a packing auxiliary takes care of placing the digits of Q_B in an appropriate way for the final addition to obtain Q. This auxiliary is also used to compute Q_C and Q_D with the inputs C and D instead of B.

```
\cs_new:Npn \__fp_div_significand_ii:wwn #1; #2;#3
  {
    \exp_after:wN \__fp_div_significand_pack:NNN
    \int_value:w \__fp_int_eval:w
      \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
      \int_value:w \__fp_int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
  }
```

(*End of definition for* `\__fp_div_significand_ii:wwn`*.*)

`\__fp_div_significand_iii:wwnnnnn`

`\__fp_div_significand_iii:wwnnnnn` ⟨y⟩ `;` ⟨E_1⟩ `;` `{`⟨E_2⟩`}` `{`⟨E_3⟩`}` `{`⟨E_4⟩`}` `{`⟨Z_1⟩`}` `{`⟨Z_2⟩`}` `{`⟨Z_3⟩`}` `{`⟨Z_4⟩`}` ⟨*sign*⟩

We compute $P \simeq 2E/Z$ by rounding $2E_1E_2/Z_1Z_2$. Note the first 0, which multiplies Q_D by 10: we later add (roughly) $5 \cdot P$, which amounts to adding $P/2 \simeq E/Z$ to Q_D, the appropriate correction from a hypothetical Q_E.

```
\cs_new:Npn \__fp_div_significand_iii:wwnnnnn #1; #2;#3#4#5 #6#7
  {
    0
    \exp_after:wN \__fp_div_significand_iv:wwnnnnnnn
    \int_value:w \__fp_int_eval:w ( 2 * #2 #3) / #6 #7 ; % <- P
      #2 ; {#3} {#4} {#5}
      {#6} {#7}
  }
```

(*End of definition for* `\__fp_div_significand_iii:wwnnnnn`*.*)

`\__fp_div_significand_iv:wwnnnnnnn`
`\__fp_div_significand_v:NNw`
`\__fp_div_significand_vi:Nw`

`\__fp_div_significand_iv:wwnnnnnnn` ⟨P⟩ `;` ⟨E_1⟩ `;` `{`⟨E_2⟩`}` `{`⟨E_3⟩`}` `{`⟨E_4⟩`}` `{`⟨Z_1⟩`}` `{`⟨Z_2⟩`}` `{`⟨Z_3⟩`}` `{`⟨Z_4⟩`}` ⟨*sign*⟩

This adds to the current expression $(10^7 + 10 \cdot Q_D)$ a contribution of $5 \cdot P + \mathrm{sign}(T)$ with $T = 2E - PZ$. This amounts to adding $P/2$ to Q_D, with an extra ⟨*rounding*⟩ digit. This ⟨*rounding*⟩ digit is 0 or 5 if T does not contribute, *i.e.,* if $0 = T = 2E - PZ$, in other words if $10^{16}A/Z$ is an integer or half-integer. Otherwise it is in the appropriate range, $[1, 4]$ or $[6, 9]$. This is precise enough for rounding purposes (in any mode).

It seems an overkill to compute T exactly as I do here, but I see no faster way right now.

Once more, we need to be careful and show that the calculation $\#1 \cdot \#6\#7$ below does not cause an overflow: naively, P can be up to 35, and $\#6\#7$ up to 10^8, but both cannot happen simultaneously. To show that things are fine, we split in two (non-disjoint) cases.

- For $P < 10$, the product obeys $P \cdot \#6\#7 < 10^8 \cdot P < 10^9$.
- For large $P \geq 3$, the rounding error on P, which is at most 1, is less than a factor of 2, hence $P \leq 4E/Z$. Also, $\#6\#7 \leq 10^8 \cdot Z$, hence $P \cdot \#6\#7 \leq 4E \cdot 10^8 < 10^9$.

Both inequalities could be made tighter if needed.

Note however that $P \cdot \#8\#9$ may overflow, since the two factors are now independent, and the result may reach $3.5 \cdot 10^9$. Thus we compute the two lower levels separately. The rest is standard, except that we use `+` as a separator (ending integer expressions explicitly). T is negative if the first character is `-`, it is positive if the first character is neither `0` nor `-`. It is also positive if the first character is `0` and second argument of `\__fp_div_significand_vi:Nw`, a sum of several terms, is also zero. Otherwise, there was an exact agreement: $T = 0$.

```
\cs_new:Npn \__fp_div_significand_iv:wwnnnnnnn #1; #2;#3#4#5 #6#7#8#9
  {
    + 5 * #1
    \exp_after:wN \__fp_div_significand_vi:Nw
    \int_value:w \__fp_int_eval:w -50 + 2*#2#3 - #1*#6#7 +
      \exp_after:wN \__fp_div_significand_v:NN
      \int_value:w \__fp_int_eval:w 499950 + 2*#4 - #1*#8 +
        \exp_after:wN \__fp_div_significand_v:NN
        \int_value:w \__fp_int_eval:w 500000 + 2*#5 - #1*#9 ;
  }
\cs_new:Npn \__fp_div_significand_v:NN #1#2 { #1#2 \__fp_int_eval_end: + }
\cs_new:Npn \__fp_div_significand_vi:Nw #1#2;
  {
    \if_meaning:w 0 #1
      \if_int_compare:w \__fp_int_eval:w #2 > 0 + 1 \fi:
    \else:
      \if_meaning:w - #1 - \else: + \fi: 1
    \fi:
    ;
  }
```

(*End of definition for* `\__fp_div_significand_iv:wwnnnnnnn`*,* `\__fp_div_significand_v:NNw`*, and* `\__fp_div_significand_vi:Nw`*.*)

`\__fp_div_significand_pack:NNN` At this stage, we are in the following situation: TEX is in the process of expanding several integer expressions, thus functions at the bottom expand before those above.

__fp_div_significand_test_o:w $10^6 + Q_A$ __fp_div_significand_pack:NNN $10^6 + Q_B$ __fp_div_significand_pack:NNN $10^6 + Q_C$ __fp_div_significand_pack:NNN $10^7 + 10 \cdot Q_D + 5 \cdot P + \varepsilon$; ⟨*sign*⟩

Here, $\varepsilon = \operatorname{sign}(T)$ is 0 in case $2E = PZ$, 1 in case $2E > PZ$, which means that P was the correct value, but not with an exact quotient, and -1 if $2E < PZ$, *i.e.*, P was an overestimate. The packing function we define now does nothing special: it removes the 10^6 and carries two digits (for the 10^5's and the 10^4's).

```
\cs_new:Npn \__fp_div_significand_pack:NNN 1 #1 #2 { + #1 #2 ; }
```

(*End of definition for* `\__fp_div_significand_pack:NNN`.)

__fp_div_significand_test_o:w

__fp_div_significand_test_o:w 1 0 ⟨*5d*⟩ ; ⟨*4d*⟩ ; ⟨*4d*⟩ ; ⟨*5d*⟩ ; ⟨*sign*⟩

The reason we know that the first two digits are 1 and 0 is that the final result is known to be between 0.1 (inclusive) and 10, hence $\widetilde{Q_A}$ (the tilde denoting the contribution from the other Q_i) is at most 99999, and $10^6 + \widetilde{Q_A} = 10 \cdots$.

It is now time to round. This depends on how many digits the final result will have.

```
\cs_new:Npn \__fp_div_significand_test_o:w 10 #1
  {
    \if_meaning:w 0 #1
      \exp_after:wN \__fp_div_significand_small_o:wwwNNNNwN
    \else:
      \exp_after:wN \__fp_div_significand_large_o:wwwNNNNwN
    \fi:
    #1
  }
```

(*End of definition for* `\__fp_div_significand_test_o:w`.)

__fp_div_significand_small_o:wwwNNNNwN

__fp_div_significand_small_o:wwwNNNNwN 0 ⟨*4d*⟩ ; ⟨*4d*⟩ ; ⟨*4d*⟩ ; ⟨*5d*⟩ ; ⟨*final sign*⟩

Standard use of the functions __fp_basics_pack_low:NNNNNw and __fp_basics_pack_high:NNNNNw. We finally get to use the ⟨*final sign*⟩ which has been sitting there for a while.

```
\cs_new:Npn \__fp_div_significand_small_o:wwwNNNNwN
    0 #1; #2; #3; #4#5#6#7#8; #9
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \int_value:w \__fp_int_eval:w 1 #1#2
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \int_value:w \__fp_int_eval:w 1 #3#4#5#6#7
        + \__fp_round:NNN #9 #7 #8
        \exp_after:wN ;
  }
```

(*End of definition for* `\__fp_div_significand_small_o:wwwNNNNwN`.)

__fp_div_significand_large_o:wwwNNNNwN

__fp_div_significand_large_o:wwwNNNNwN ⟨*5d*⟩ ; ⟨*4d*⟩ ; ⟨*4d*⟩ ; ⟨*5d*⟩ ; ⟨*sign*⟩

We know that the final result cannot reach 10, hence 1#1#2, together with contributions from the level below, cannot reach $2 \cdot 10^9$. For rounding, we build the ⟨*rounding digit*⟩ from the last two of our 18 digits.

```
\cs_new:Npn \__fp_div_significand_large_o:wwwNNNNwN
```

```
    #1; #2; #3; #4#5#6#7#8; #9
  {
    + 1
    \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw
    \int_value:w \__fp_int_eval:w 1 #1 #2
      \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
      \int_value:w \__fp_int_eval:w 1 #3 #4 #5 #6 +
        \exp_after:wN \__fp_round:NNN
        \exp_after:wN #9
        \exp_after:wN #6
        \int_value:w \__fp_round_digit:Nw #7 #8 ;
      \exp_after:wN ;
  }
```

(*End of definition for* `\__fp_div_significand_large_o:wwwNNNNwN`.)

74.4 Square root

`\__fp_sqrt_o:w`

Zeros are unchanged: $\sqrt{-0} = -0$ and $\sqrt{+0} = +0$. Negative numbers (other than -0) have no real square root. Positive infinity, and `nan`, are unchanged. Finally, for normal positive numbers, there is some work to do.

```
\cs_new:Npn \__fp_sqrt_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_meaning:w 0 #2 \__fp_case_return_same_o:w \fi:
    \if_meaning:w 2 #3
      \__fp_case_use:nw { \__fp_invalid_operation_o:nw { sqrt } }
    \fi:
    \if_meaning:w 1 #2 \else: \__fp_case_return_same_o:w \fi:
    \__fp_sqrt_npos_o:w
    \s__fp \__fp_chk:w #2 #3 #4;
  }
```

(*End of definition for* `\__fp_sqrt_o:w`.)

`\__fp_sqrt_npos_o:w`
`\__fp_sqrt_npos_auxi_o:wwnnN`
`\__fp_sqrt_npos_auxii_o:wNNNNNNNN`

Prepare `\__fp_sanitize:Nw` to receive the final sign `0` (the result is always positive) and the exponent, equal to half of the exponent `#1` of the argument. If the exponent `#1` is even, find a first approximation of the square root of the significand $10^8 a_1 + a_2 = 10^8$`#2#3`+`#4#5` through Newton's method, starting at $x = 57234133 \simeq 10^{7.75}$. Otherwise, first shift the significand of the argument by one digit, getting $a_1' \in [10^6, 10^7)$ instead of $[10^7, 10^8)$, then use Newton's method starting at $17782794 \simeq 10^{7.25}$.

```
\cs_new:Npn \__fp_sqrt_npos_o:w \s__fp \__fp_chk:w 1 0 #1#2#3#4#5;
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN 0
    \int_value:w \__fp_int_eval:w
      \if_int_odd:w #1 \exp_stop_f:
        \exp_after:wN \__fp_sqrt_npos_auxi_o:wwnnN
      \fi:
      #1 / 2
      \__fp_sqrt_Newton_o:wwn 56234133; 0; {#2#3} {#4#5} 0
  }
\cs_new:Npn \__fp_sqrt_npos_auxi_o:wwnnN #1 / 2 #2; 0; #3#4#5
  {
```

```
    ( #1 + 1 ) / 2
    \__fp_pack_eight:wNNNNNNNN
    \__fp_sqrt_npos_auxii_o:wNNNNNNNN
    ;
    0 #3 #4
  }
\cs_new:Npn \__fp_sqrt_npos_auxii_o:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
  { \__fp_sqrt_Newton_o:wwn 17782794; 0; {#1} {#2#3#4#5#6#7#8#9} }
```

(*End of definition for* `\__fp_sqrt_npos_o:w`*,* `\__fp_sqrt_npos_auxi_o:wwnnN`*, and* `\__fp_sqrt_npos_auxii_o:wNNNNNNNN`*.*)

`\__fp_sqrt_Newton_o:wwn`

Newton's method maps $x \mapsto \big[(x + [10^8 a_1/x])/2\big]$ in each iteration, where $[b/c]$ denotes ε-TEX's division. This division rounds the real number b/c to the closest integer, rounding ties away from zero, hence when c is even, $b/c - 1/2 + 1/c \le [b/c] \le b/c + 1/2$ and when c is odd, $b/c - 1/2 + 1/(2c) \le [b/c] \le b/c + 1/2 - 1/(2c)$. For all c, $b/c - 1/2 + 1/(2c) \le [b/c] \le b/c + 1/2$.

Let us prove that the method converges when implemented with ε-TEX integer division, for any $10^6 \le a_1 < 10^8$ and starting value $10^6 \le x < 10^8$. Using the inequalities above and the arithmetic–geometric inequality $(x+t)/2 \ge \sqrt{xt}$ for $t = 10^8 a_1/x$, we find

$$x' = \left[\frac{x + [10^8 a_1/x]}{2}\right] \ge \frac{x + 10^8 a_1/x - 1/2 + 1/(2x)}{2} \ge \sqrt{10^8 a_1} - \frac{1}{4} + \frac{1}{4x} \,.$$

After any step of iteration, we thus have $\delta = x - \sqrt{10^8 a_1} \ge -0.25 + 0.25 \cdot 10^{-8}$. The new difference $\delta' = x' - \sqrt{10^8 a_1}$ after one step is bounded above as

$$x' - \sqrt{10^8 a_1} \le \frac{x + 10^8 a_1/x + 1/2}{2} + \frac{1}{2} - \sqrt{10^8 a_1} \le \frac{\delta}{2} \frac{\delta}{\sqrt{10^8 a_1} + \delta} + \frac{3}{4} \,.$$

For $\delta > 3/2$, this last expression is $\le \delta/2 + 3/4 < \delta$, hence δ decreases at each step: since all x are integers, δ must reach a value $-1/4 < \delta \le 3/2$. In this range of values, we get $\delta' \le \frac{3}{4} \frac{3}{2\sqrt{10^8 a_1}} + \frac{3}{4} \le 0.75 + 1.125 \cdot 10^{-7}$. We deduce that the difference $\delta = x - \sqrt{10^8 a_1}$ eventually reaches a value in the interval $[-0.25 + 0.25 \cdot 10^{-8}, 0.75 + 11.25 \cdot 10^{-8}]$, whose width is $1 + 11 \cdot 10^{-8}$. The corresponding interval for x may contain two integers, hence x might oscillate between those two values.

However, the fact that $x \mapsto x - 1$ and $x - 1 \mapsto x$ puts stronger constraints, which are not compatible: the first implies

$$x + [10^8 a_1/x] \le 2x - 2$$

hence $10^8 a_1/x \le x - 3/2$, while the second implies

$$x - 1 + [10^8 a_1/(x-1)] \ge 2x - 1$$

hence $10^8 a_1/(x-1) \ge x - 1/2$. Combining the two inequalities yields $x^2 - 3x/2 \ge 10^8 a_1 \ge x - 3x/2 + 1/2$, which cannot hold. Therefore, the iteration always converges to a single integer x. To stop the iteration when two consecutive results are equal, the function `\__fp_sqrt_Newton_o:wwn` receives the newly computed result as `#1`, the previous result as `#2`, and a_1 as `#3`. Note that ε-TEX combines the computation of a multiplication and a following division, thus avoiding overflow in `#3 * 100000000 / #1`. In any case, the result is within $[10^7, 10^8]$.

```
\cs_new:Npn \__fp_sqrt_Newton_o:wwn #1; #2; #3
  {
    \if_int_compare:w #1 = #2 \exp_stop_f:
      \exp_after:wN \__fp_sqrt_auxi_o:NNNNwnnN
      \int_value:w \__fp_int_eval:w 9999 9999 +
        \exp_after:wN \__fp_use_none_until_s:w
    \fi:
    \exp_after:wN \__fp_sqrt_Newton_o:wwn
    \int_value:w \__fp_int_eval:w (#1 + #3 * 1 0000 0000 / #1) / 2 ;
    #1; {#3}
  }
```

(*End of definition for* `\__fp_sqrt_Newton_o:wwn`.)

`\__fp_sqrt_auxi_o:NNNNwnnN` This function is followed by $10^8 + x - 1$, which has 9 digits starting with 1, then `;` `{`$\langle a_1\rangle$`}` `{`$\langle a_2\rangle$`}` $\langle$`a'`$\rangle$. Here, $x \simeq \sqrt{10^8 a_1}$ and we want to estimate the square root of $a = 10^{-8}a_1 + 10^{-16}a_2 + 10^{-17}a'$. We set up an initial underestimate

$$y = (x-1)10^{-8} + 0.2499998875 \cdot 10^{-8} \lesssim \sqrt{a}\,.$$

From the inequalities shown earlier, we know that $y \leq \sqrt{10^{-8}a_1} \leq \sqrt{a}$ and that $\sqrt{10^{-8}a_1} \leq y + 10^{-8} + 11 \cdot 10^{-16}$ hence (using $0.1 \leq y \leq \sqrt{a} \leq 1$)

$$a - y^2 \leq 10^{-8}a_1 + 10^{-8} - y^2 \leq (y + 10^{-8} + 11 \cdot 10^{-16})^2 - y^2 + 10^{-8} < 3.2 \cdot 10^{-8}\,,$$

and $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y) \leq 16 \cdot 10^{-8}$. Next, `\__fp_sqrt_auxii_o:NnnnnnnnN` is called several times to get closer and closer underestimates of $\sqrt{a}$. By construction, the underestimates y are always increasing, $a - y^2 < 3.2 \cdot 10^{-8}$ for all. Also, $y < 1$.

```
\cs_new:Npn \__fp_sqrt_auxi_o:NNNNwnnN 1 #1#2#3#4#5;
  {
    \__fp_sqrt_auxii_o:NnnnnnnnN
      \__fp_sqrt_auxiii_o:wnnnnnnnn
      {#1#2#3#4} {#5} {2499} {9988} {7500}
  }
```

(*End of definition for* `\__fp_sqrt_auxi_o:NNNNwnnN`.)

`\__fp_sqrt_auxii_o:NnnnnnnnN` This receives a continuation function `#1`, then five blocks of 4 digits for y, then two 8-digit blocks and a single digit for a. A common estimate of $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y)$ is $(a - y^2)/(2y)$, which leads to alternating overestimates and underestimates. We tweak this, to only work with underestimates (no need then to worry about signs in the computation). Each step finds the largest integer $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then computes the integer (with ε-TeX's rounding division)

$$10^{4j} z = \Bigl[\bigl(\lfloor 10^{4j}(a - y^2)\rfloor - 257\bigr) \cdot (0.5 \cdot 10^8) \Bigm/ \lfloor 10^8 y + 1\rfloor\Bigr]\,.$$

The choice of j ensures that $10^{4j} z < 2 \cdot 10^8 \cdot 0.5 \cdot 10^8 / 10^7 = 10^9$, thus $10^9 + 10^{4j} z$ has exactly 10 digits, does not overflow TeX's integer range, and starts with 1. Incidentally, since all $a - y^2 \leq 3.2 \cdot 10^{-8}$, we know that $j \geq 3$.

Let us show that z is an underestimate of $\sqrt{a} - y$. On the one hand, $\sqrt{a} - y \leq 16 \cdot 10^{-8}$ because this holds for the initial y and values of y can only increase. On the other hand, the choice of j implies that $\sqrt{a} - y \leq 5(\sqrt{a} + y)(\sqrt{a} - y) = 5(a - y^2) < 10^{9-4j}$. For $j = 3$, the first bound is better, while for larger j, the second bound is better. For all $j \in [3, 6]$, we find $\sqrt{a} - y < 16 \cdot 10^{-2j}$. From this, we deduce that

$$10^{4j}(\sqrt{a} - y) = \frac{10^{4j}\bigl(a - y^2 - (\sqrt{a} - y)^2\bigr)}{2y} \geq \frac{\lfloor 10^{4j}(a - y^2) \rfloor - 257}{2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor} + \frac{1}{2}$$

where we have replaced the bound $10^{4j}(16 \cdot 10^{-2j}) = 256$ by 257 and extracted the corresponding term $1/\bigl(2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor\bigr) \geq 1/2$. Given that ε-TEX's integer division obeys $[b/c] \leq b/c + 1/2$, we deduce that $10^{4j} z \leq 10^{4j}(\sqrt{a} - y)$, hence $y + z \leq \sqrt{a}$ is an underestimate of $\sqrt{a}$, as claimed. One implementation detail: because the computation involves `-#4*#4 - 2*#3*#5 - 2*#2*#6` which may be as low as $-5 \cdot 10^8$, we need to use the `pack_big` functions, and the `big` shifts.

```
\cs_new:Npn \__fp_sqrt_auxii_o:NnnnnnnnN #1 #2#3#4#5#6 #7#8#9
  {
    \exp_after:wN #1
    \int_value:w \__fp_int_eval:w \c__fp_big_leading_shift_int
      + #7 - #2 * #2
      \exp_after:wN \__fp_pack_big:NNNNNNw
      \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
        - 2 * #2 * #3
        \exp_after:wN \__fp_pack_big:NNNNNNw
        \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
          + #8 - #3 * #3 - 2 * #2 * #4
          \exp_after:wN \__fp_pack_big:NNNNNNw
          \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
            - 2 * #3 * #4 - 2 * #2 * #5
            \exp_after:wN \__fp_pack_big:NNNNNNw
            \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
              + #9 000 0000 - #4 * #4 - 2 * #3 * #5 - 2 * #2 * #6
              \exp_after:wN \__fp_pack_big:NNNNNNw
              \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
                - 2 * #4 * #5 - 2 * #3 * #6
                \exp_after:wN \__fp_pack_big:NNNNNNw
                \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
                  - #5 * #5 - 2 * #4 * #6
                  \exp_after:wN \__fp_pack_big:NNNNNNw
                  \int_value:w \__fp_int_eval:w
                    \c__fp_big_middle_shift_int
                    - 2 * #5 * #6
                    \exp_after:wN \__fp_pack_big:NNNNNNw
                    \int_value:w \__fp_int_eval:w
                      \c__fp_big_trailing_shift_int
                      - #6 * #6 ;
    % (
    - 257 ) * 5000 0000 / (#2#3 + 1) + 10 0000 0000 ;
    {#2}{#3}{#4}{#5}{#6} {#7}{#8}#9
  }
```

(*End of definition for* `\__fp_sqrt_auxii_o:NnnnnnnnN`.)

`\__fp_sqrt_auxiii_o:wnnnnnnnn`
`\__fp_sqrt_auxiv_o:NNNNNw`
`\__fp_sqrt_auxv_o:NNNNNw`
`\__fp_sqrt_auxvi_o:NNNNNw`
`\__fp_sqrt_auxvii_o:NNNNNw`

We receive here the difference $a - y^2 = d = \sum_i d_i \cdot 10^{-4i}$, as $\langle d_2 \rangle$ `;` `{`$\langle d_3 \rangle$`}` … `{`$\langle d_{10} \rangle$`}`, where each block has 4 digits, except $\langle d_2 \rangle$. This function finds the largest $j \leq 6$ such that $10^{4j}(a - y^2) < 2 \cdot 10^8$, then leaves an open parenthesis and the integer $\lfloor 10^{4j}(a - y^2) \rfloor$

in an integer expression. The closing parenthesis is provided by the caller `\__fp_sqrt_auxii_o:NnnnnnnnN`, which completes the expression

$$10^{4j}z = \left[\left(\lfloor 10^{4j}(a-y^2)\rfloor - 257\right)\cdot(0.5\cdot 10^8)\Bigm/\lfloor 10^8 y+1\rfloor\right]$$

for an estimate of $10^{4j}(\sqrt{a}-y)$. If $d_2 \geq 2$, $j=3$ and the `auxiv` auxiliary receives $10^{12}z$. If $d_2 \leq 1$ but $10^4 d_2 + d_3 \geq 2$, $j=4$ and the `auxv` auxiliary is called, and receives $10^{16}z$, and so on. In all those cases, the `auxviii` auxiliary is set up to add z to y, then go back to the `auxii` step with continuation `auxiii` (the function we are currently describing). The maximum value of j is 6, regardless of whether $10^{12}d_2 + 10^8 d_3 + 10^4 d_4 + d_5 \geq 1$. In this last case, we detect when $10^{24}z < 10^7$, which essentially means $\sqrt{a}-y \lesssim 10^{-17}$: once this threshold is reached, there is enough information to find the correctly rounded $\sqrt{a}$ with only one more call to `\__fp_sqrt_auxii_o:NnnnnnnnN`. Note that the iteration cannot be stuck before reaching $j=6$, because for $j<6$, one has $2\cdot 10^8 \leq 10^{4(j+1)}(a-y^2)$, hence

$$10^{4j}z \geq \frac{(20000-257)(0.5\cdot 10^8)}{\lfloor 10^8 y+1\rfloor} \geq (20000-257)\cdot 0.5 > 0\,.$$

```
\cs_new:Npn \__fp_sqrt_auxiii_o:wnnnnnnnn
    #1; #2#3#4#5#6#7#8#9
  {
    \if_int_compare:w #1 > \c_one_int
      \exp_after:wN \__fp_sqrt_auxiv_o:NNNNNw
      \int_value:w \__fp_int_eval:w (#1#2 %)
    \else:
      \if_int_compare:w #1#2 > \c_one_int
        \exp_after:wN \__fp_sqrt_auxv_o:NNNNNw
        \int_value:w \__fp_int_eval:w (#1#2#3 %)
      \else:
        \if_int_compare:w #1#2#3 > \c_one_int
          \exp_after:wN \__fp_sqrt_auxvi_o:NNNNNw
          \int_value:w \__fp_int_eval:w (#1#2#3#4 %)
        \else:
          \exp_after:wN \__fp_sqrt_auxvii_o:NNNNNw
          \int_value:w \__fp_int_eval:w (#1#2#3#4#5 %)
        \fi:
      \fi:
    \fi:
  }
\cs_new:Npn \__fp_sqrt_auxiv_o:NNNNNw 1#1#2#3#4#5#6;
  { \__fp_sqrt_auxviii_o:nnnnnnn {#1#2#3#4#5#6} {00000000} }
\cs_new:Npn \__fp_sqrt_auxv_o:NNNNNw 1#1#2#3#4#5#6;
  { \__fp_sqrt_auxviii_o:nnnnnnn {000#1#2#3#4#5} {#60000} }
\cs_new:Npn \__fp_sqrt_auxvi_o:NNNNNw 1#1#2#3#4#5#6;
  { \__fp_sqrt_auxviii_o:nnnnnnn {0000000#1} {#2#3#4#5#6} }
\cs_new:Npn \__fp_sqrt_auxvii_o:NNNNNw 1#1#2#3#4#5#6;
  {
    \if_int_compare:w #1#2 = \c_zero_int
      \exp_after:wN \__fp_sqrt_auxx_o:Nnnnnnnn
    \fi:
    \__fp_sqrt_auxviii_o:nnnnnnn {00000000} {000#1#2#3#4#5}
  }
```

(*End of definition for* `\__fp_sqrt_auxiii_o:wnnnnnnnn` *and others.*)

__fp_sqrt_auxviii_o:nnnnnnn
__fp_sqrt_auxix_o:wnwnw

Simply add the two 8-digit blocks of z, aligned to the last four of the five 4-digit blocks of y, then call the auxii auxiliary to evaluate $y'^2 = (y+z)^2$.

```
\cs_new:Npn \__fp_sqrt_auxviii_o:nnnnnnn #1#2 #3#4#5#6#7
  {
    \exp_after:wN \__fp_sqrt_auxix_o:wnwnw
    \int_value:w \__fp_int_eval:w #3
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \int_value:w \__fp_int_eval:w #1 + 1#4#5
        \exp_after:wN \__fp_basics_pack_low:NNNNNw
        \int_value:w \__fp_int_eval:w #2 + 1#6#7 ;
  }
\cs_new:Npn \__fp_sqrt_auxix_o:wnwnw #1; #2#3; #4#5;
  {
    \__fp_sqrt_auxii_o:NnnnnnnnN
      \__fp_sqrt_auxiii_o:wnnnnnnnn {#1}{#2}{#3}{#4}{#5}
  }
```

(*End of definition for* __fp_sqrt_auxviii_o:nnnnnnn *and* __fp_sqrt_auxix_o:wnwnw.)

__fp_sqrt_auxx_o:Nnnnnnnn
__fp_sqrt_auxxi_o:wwnnN

At this stage, $j = 6$ and $10^{24}z < 10^7$, hence

$$10^7 + 1/2 > 10^{24}z + 1/2 \geq \left(10^{24}(a-y^2) - 258\right) \cdot (0.5 \cdot 10^8) \Big/ (10^8 y + 1)\,,$$

then $10^{24}(a-y^2) - 258 < 2(10^7+1/2)(y+10^{-8})$, and

$$10^{24}(a-y^2) < (10^7 + 1290.5)(1+10^{-8}/y)(2y) < (10^7+1290.5)(1+10^{-7})(y+\sqrt{a})\,,$$

which finally implies $0 \leq \sqrt{a} - y < 0.2 \cdot 10^{-16}$. In particular, y is an underestimate of $\sqrt{a}$ and $y + 0.5 \cdot 10^{-16}$ is a (strict) overestimate. There is at exactly one multiple m of $0.5 \cdot 10^{-16}$ in the interval $[y, y+0.5\cdot 10^{-16})$. If $m^2 > a$, then the square root is inexact and is obtained by rounding $m - \epsilon$ to a multiple of 10^{-16} (the precise shift $0 < \epsilon < 0.5 \cdot 10^{-16}$ is irrelevant for rounding). If $m^2 = a$ then the square root is exactly m, and there is no rounding. If $m^2 < a$ then we round $m + \epsilon$. For now, discard a few irrelevant arguments #1, #2, #3, and find the multiple of $0.5 \cdot 10^{-16}$ within $[y, y+0.5\cdot 10^{-16})$; rather, only the last 4 digits #8 of y are considered, and we do not perform any carry yet. The auxxi auxiliary sets up auxii with a continuation function auxxii instead of auxiii as before. To prevent auxii from giving a negative results $a - m^2$, we compute $a + 10^{-16} - m^2$ instead, always positive since $m < \sqrt{a} + 0.5 \cdot 10^{-16}$ and $a \leq 1 - 10^{-16}$.

```
\cs_new:Npn \__fp_sqrt_auxx_o:Nnnnnnnn #1#2#3 #4#5#6#7#8
  {
    \exp_after:wN \__fp_sqrt_auxxi_o:wwnnN
    \int_value:w \__fp_int_eval:w
      (#8 + 2499) / 5000 * 5000 ;
      {#4} {#5} {#6} {#7} ;
  }
\cs_new:Npn \__fp_sqrt_auxxi_o:wwnnN #1; #2; #3#4#5
  {
    \__fp_sqrt_auxii_o:NnnnnnnnN
      \__fp_sqrt_auxxii_o:nnnnnnnnw
      #2 {#1}
      {#3} { #4 + 1 } #5
  }
```

(*End of definition for* `\__fp_sqrt_auxx_o:Nnnnnnnn` *and* `\__fp_sqrt_auxxi_o:wwnnN`.)

`\__fp_sqrt_auxxii_o:nnnnnnnnw`
`\__fp_sqrt_auxxiii_o:w`

The difference $0 \leq a + 10^{-16} - m^2 \leq 10^{-16} + (\sqrt{a} - m)(\sqrt{a} + m) \leq 2 \cdot 10^{-16}$ was just computed: its first 8 digits vanish, as do the next four, `#1`, and most of the following four, `#2`. The guess m is an overestimate if $a + 10^{-16} - m^2 < 10^{-16}$, that is, `#1#2` vanishes. Otherwise it is an underestimate, unless $a + 10^{-16} - m^2 = 10^{-16}$ exactly. For an underestimate, call the `auxxiv` function with argument 9998. For an exact result call it with 9999, and for an overestimate call it with 10000.

```
\cs_new:Npn \__fp_sqrt_auxxii_o:nnnnnnnnw 0; #1#2#3#4#5#6#7#8 #9;
  {
    \if_int_compare:w #1#2 > \c_zero_int
      \if_int_compare:w #1#2 = \c_one_int
        \if_int_compare:w #3#4 = \c_zero_int
          \if_int_compare:w #5#6 = \c_zero_int
            \if_int_compare:w #7#8 = \c_zero_int
              \__fp_sqrt_auxxiii_o:w
            \fi:
          \fi:
        \fi:
      \fi:
      \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnN
      \int_value:w 9998
    \else:
      \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnN
      \int_value:w 10000
    \fi:
    ;
  }
\cs_new:Npn \__fp_sqrt_auxxiii_o:w \fi: \fi: \fi: \fi: #1 \fi: ;
  {
    \fi: \fi: \fi: \fi: \fi:
    \__fp_sqrt_auxxiv_o:wnnnnnnnN 9999 ;
  }
```

(*End of definition for* `\__fp_sqrt_auxxii_o:nnnnnnnnw` *and* `\__fp_sqrt_auxxiii_o:w`.)

`\__fp_sqrt_auxxiv_o:wnnnnnnnN`

This receives 9998, 9999 or 10000 as `#1` when m is an underestimate, exact, or an overestimate, respectively. Then comes m as five blocks of 4 digits, but where the last block `#6` may be 0, 5000, or 10000. In the latter case, we need to add a carry, unless m is an overestimate (`#1` is then 10000). Then comes a as three arguments. Rounding is done by `\__fp_round:NNN`, whose first argument is the final sign 0 (square roots are positive). We fake its second argument. It should be the last digit kept, but this is only used when ties are "rounded to even", and only when the result is exactly half-way between two representable numbers rational square roots of numbers with 16 significant digits have: this situation never arises for the square root, as any exact square root of a 16 digit number has at most 8 significant digits. Finally, the last argument is the next digit, possibly shifted by 1 when there are further nonzero digits. This is achieved by `\__fp_round_digit:Nw`, which receives (after removal of the 10000's digit) one of 0000, 0001, 4999, 5000, 5001, or 9999, which it converts to 0, 1, 4, 5, 6, and 9, respectively.

```
\cs_new:Npn \__fp_sqrt_auxxiv_o:wnnnnnnnN #1; #2#3#4#5#6 #7#8#9
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \int_value:w \__fp_int_eval:w 1 0000 0000 + #2#3
```

```
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \int_value:w \__fp_int_eval:w 1 0000 0000
        + #4#5
        \if_int_compare:w #6 > #1 \exp_stop_f: + 1 \fi:
        + \exp_after:wN \__fp_round:NNN
          \exp_after:wN 0
          \exp_after:wN 0
          \int_value:w
            \exp_after:wN \use_i:nn
            \exp_after:wN \__fp_round_digit:Nw
            \int_value:w \__fp_int_eval:w #6 + 19999 - #1 ;
    \exp_after:wN ;
  }
```

(*End of definition for* `\__fp_sqrt_auxxiv_o:wnnnnnnnN`.)

74.5 About the sign and exponent

`\__fp_logb_o:w`
`\__fp_logb_aux_o:w`

The exponent of a normal number is its ⟨*exponent*⟩ minus one.

```
\cs_new:Npn \__fp_logb_o:w ? \s__fp \__fp_chk:w #1#2; @
  {
    \if_case:w #1 \exp_stop_f:
           \__fp_case_use:nw
             { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { logb } }
    \or:   \exp_after:wN \__fp_logb_aux_o:w
    \or:   \__fp_case_return_o:Nw \c_inf_fp
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #1 #2;
  }
\cs_new:Npn \__fp_logb_aux_o:w \s__fp \__fp_chk:w #1 #2 #3 #4 ;
  {
    \exp_after:wN \__fp_parse:n \exp_after:wN
      { \int_value:w \int_eval:w #3 - 1 \exp_after:wN }
  }
```

(*End of definition for* `\__fp_logb_o:w` *and* `\__fp_logb_aux_o:w`.)

`\__fp_sign_o:w`
`\__fp_sign_aux_o:w`

Find the sign of the floating point: `nan`, `+0`, `-0`, `+1` or `-1`.

```
\cs_new:Npn \__fp_sign_o:w ? \s__fp \__fp_chk:w #1#2; @
  {
    \if_case:w #1 \exp_stop_f:
           \__fp_case_return_same_o:w
    \or:   \exp_after:wN \__fp_sign_aux_o:w
    \or:   \exp_after:wN \__fp_sign_aux_o:w
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #1 #2;
  }
\cs_new:Npn \__fp_sign_aux_o:w \s__fp \__fp_chk:w #1 #2 #3 ;
  { \exp_after:wN \__fp_set_sign_o:w \exp_after:wN #2 \c_one_fp @ }
```

(*End of definition for* `\__fp_sign_o:w` *and* `\__fp_sign_aux_o:w`.)

__fp_set_sign_o:w This function is used for the unary minus and for abs. It leaves the sign of nan invariant, turns negative numbers (sign 2) to positive numbers (sign 0) and positive numbers (sign 0) to positive or negative numbers depending on #1. It also expands after itself in the input stream, just like __fp_+_o:ww.

```
\cs_new:Npn \__fp_set_sign_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \exp_after:wN \__fp_exp_after_o:w
    \exp_after:wN \s__fp
    \exp_after:wN \__fp_chk:w
    \exp_after:wN #2
    \int_value:w
      \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
    #4;
  }
```

(*End of definition for* __fp_set_sign_o:w.)

74.6 Operations on tuples

__fp_tuple_set_sign_o:w
__fp_tuple_set_sign_aux_o:Nnw
__fp_tuple_set_sign_aux_o:w

Two cases: abs(⟨*tuple*⟩) for which #1 is 0 (invalid for tuples) and -⟨*tuple*⟩ for which #1 is 2. In that case, map over all items in the tuple an auxiliary that dispatches to the type-appropriate sign-flipping function.

```
\cs_new:Npn \__fp_tuple_set_sign_o:w #1#2 @
  {
    \if_meaning:w 2 #1
      \exp_after:wN \__fp_tuple_set_sign_aux_o:Nnw
    \fi:
    \__fp_invalid_operation_o:nw { abs }
    #2
  }
\cs_new:Npn \__fp_tuple_set_sign_aux_o:Nnw #1#2
  { \__fp_tuple_map_o:nw \__fp_tuple_set_sign_aux_o:w }
\cs_new:Npn \__fp_tuple_set_sign_aux_o:w #1#2 ;
  {
    \__fp_change_func_type:NNN #1 \__fp_set_sign_o:w
      \__fp_parse_apply_unary_error:NNw
    2 #1 #2 ; @
  }
```

(*End of definition for* __fp_tuple_set_sign_o:w, __fp_tuple_set_sign_aux_o:Nnw, *and* __fp_tuple_set_sign_aux_o:w.)

__fp_*_tuple_o:ww
__fp_tuple_*_o:ww
__fp_tuple_/_o:ww

For ⟨*number*⟩*⟨*tuple*⟩ and ⟨*tuple*⟩*⟨*number*⟩ and ⟨*tuple*⟩/⟨*number*⟩, loop through the ⟨*tuple*⟩ some code that multiplies or divides by the appropriate ⟨*number*⟩. Importantly we need to dispatch according to the type, and we make sure to apply the operator in the correct order.

```
\cs_new:cpn { __fp_*_tuple_o:ww } #1 ;
  { \__fp_tuple_map_o:nw { \__fp_binary_type_o:Nww * #1 ; } }
\cs_new:cpn { __fp_tuple_*_o:ww } #1 ; #2 ;
  { \__fp_tuple_map_o:nw { \__fp_binary_rev_type_o:Nww * #2 ; } #1 ; }
\cs_new:cpn { __fp_tuple_/_o:ww } #1 ; #2 ;
  { \__fp_tuple_map_o:nw { \__fp_binary_rev_type_o:Nww / #2 ; } #1 ; }
```

(*End of definition for* `\__fp_*_tuple_o:ww`*,* `\__fp_tuple_*_o:ww`*, and* `\__fp_tuple_/_o:ww`*.*)

`\__fp_tuple_+_tuple_o:ww`
`\__fp_tuple_-_tuple_o:ww`

Check the two tuples have the same number of items and map through these a helper that dispatches appropriately depending on the types. This means `(1,2)+((1,1),2)` gives `(nan,4)`.

```
\cs_set_protected:Npn \__fp_tmp:w #1
  {
    \cs_new:cpn { __fp_tuple_#1_tuple_o:ww }
        \s__fp_tuple \__fp_tuple_chk:w ##1 ;
        \s__fp_tuple \__fp_tuple_chk:w ##2 ;
      {
        \int_compare:nNnTF
          { \__fp_array_count:n {##1} } = { \__fp_array_count:n {##2} }
          { \__fp_tuple_mapthread_o:nww { \__fp_binary_type_o:Nww #1 } }
          { \__fp_invalid_operation_o:nww #1 }
        \s__fp_tuple \__fp_tuple_chk:w {##1} ;
        \s__fp_tuple \__fp_tuple_chk:w {##2} ;
      }
  }
\__fp_tmp:w +
\__fp_tmp:w -
```

(*End of definition for* `\__fp_tuple_+_tuple_o:ww` *and* `\__fp_tuple_-_tuple_o:ww`*.*)

```
</package>
```

Chapter 75

l3fp-extended implementation

```
⟨*package⟩
⟨@@=fp⟩
```

75.1 Description of fixed point numbers

This module provides a few functions to manipulate positive floating point numbers with extended precision (24 digits), but mostly provides functions for fixed-point numbers with this precision (24 digits). Those are used in the computation of Taylor series for the logarithm, exponential, and trigonometric functions. Since we eventually only care about the 16 first digits of the final result, some of the calculations are not performed with the full 24-digit precision. In other words, the last two blocks of each fixed point number may be wrong as long as the error is small enough to be rounded away when converting back to a floating point number. The fixed point numbers are expressed as

{⟨a_1⟩} {⟨a_2⟩} {⟨a_3⟩} {⟨a_4⟩} {⟨a_5⟩} {⟨a_6⟩} ;

where each ⟨a_i⟩ is exactly 4 digits (ranging from 0000 to 9999), except ⟨a_1⟩, which may be any "not-too-large" non-negative integer, with or without leading zeros. Here, "not-too-large" depends on the specific function (see the corresponding comments for details). Checking for overflow is the responsibility of the code calling those functions. The fixed point number a corresponding to the representation above is $a = \sum_{i=1}^{6} \langle a_i \rangle \cdot 10^{-4i}$.

Most functions we define here have the form

`\__fp_fixed_`⟨*calculation*⟩`:wwn` ⟨*operand*$_1$⟩ `;` ⟨*operand*$_2$⟩ `;` `{`⟨*continuation*⟩`}`

They perform the ⟨*calculation*⟩ on the two ⟨*operands*⟩, then feed the result (6 brace groups followed by a semicolon) to the ⟨*continuation*⟩, responsible for the next step of the calculation. Some functions only accept an N-type ⟨*continuation*⟩. This allows constructions such as

```
\__fp_fixed_add:wwn ⟨X1⟩ ; ⟨X2⟩ ;
\__fp_fixed_mul:wwn ⟨X3⟩ ;
\__fp_fixed_add:wwn ⟨X4⟩ ;
```

to compute $(X_1 + X_2) \cdot X_3 + X_4$. This turns out to be very appropriate for computing continued fractions and Taylor series.

At the end of the calculation, the result is turned back to a floating point number using `\__fp_fixed_to_float_o:wN`. This function has to change the exponent of the floating point number: it must be used after starting an integer expression for the overall exponent of the result.

75.2 Helpers for numbers with extended precision

`\c__fp_one_fixed_tl` The fixed-point number 1, used in l3fp-expo.

```
\tl_const:Nn \c__fp_one_fixed_tl
  { {10000} {0000} {0000} {0000} {0000} {0000} ; }
```

(*End of definition for* `\c__fp_one_fixed_tl`.)

`\__fp_fixed_continue:wn` This function simply calls the next function.

```
\cs_new:Npn \__fp_fixed_continue:wn #1; #2 { #2 #1; }
```

(*End of definition for* `\__fp_fixed_continue:wn`.)

`\__fp_fixed_add_one:wN`

`\__fp_fixed_add_one:wN` ⟨a⟩ ; ⟨*continuation*⟩

This function adds 1 to the fixed point ⟨a⟩, by changing a_1 to $10000 + a_1$, then calls the ⟨*continuation*⟩. This requires $a_1 + 10000 < 2^{31}$.

```
\cs_new:Npn \__fp_fixed_add_one:wN #1#2; #3
  {
    \exp_after:wN #3 \exp_after:wN
      { \int_value:w \__fp_int_eval:w \c__fp_myriad_int + #1 } #2 ;
  }
```

(*End of definition for* `\__fp_fixed_add_one:wN`.)

`\__fp_fixed_div_myriad:wn` Divide a fixed point number by 10000. This is a little bit more subtle than just removing the last group and adding a leading group of zeros: the first group #1 may have any number of digits, and we must split #1 into the new first group and a second group of exactly 4 digits. The choice of shifts allows #1 to be in the range $[0, 5 \cdot 10^8 - 1]$.

```
\cs_new:Npn \__fp_fixed_div_myriad:wn #1#2#3#4#5#6;
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
      \exp_after:wN \__fp_pack:NNNNNw
      \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
        + #1 ; {#2}{#3}{#4}{#5};
  }
```

(*End of definition for* `\__fp_fixed_div_myriad:wn`.)

`\__fp_fixed_mul_after:wwn` The fixed point operations which involve multiplication end by calling this auxiliary. It braces the last block of digits, and places the ⟨*continuation*⟩ #3 in front.

```
\cs_new:Npn \__fp_fixed_mul_after:wwn #1; #2; #3 { #3 {#1} #2; }
```

(*End of definition for* `\__fp_fixed_mul_after:wwn`.)

75.3 Multiplying a fixed point number by a short one

__fp_fixed_mul_short:wwn

`\__fp_fixed_mul_short:wwn`
`{⟨a1⟩} {⟨a2⟩} {⟨a3⟩} {⟨a4⟩} {⟨a5⟩} {⟨a6⟩} ;`
`{⟨b0⟩} {⟨b1⟩} {⟨b2⟩} ; {⟨continuation⟩}`

Computes the product $c = ab$ of $a = \sum_i \langle a_i\rangle 10^{-4i}$ and $b = \sum_i \langle b_i\rangle 10^{-4i}$, rounds it to the closest multiple of 10^{-24}, and leaves ⟨*continuation*⟩ {⟨c_1⟩} ... {⟨c_6⟩} ; in the input stream, where each of the ⟨c_i⟩ are blocks of 4 digits, except ⟨c_1⟩, which is any TEX integer. Note that indices for ⟨*b*⟩ start at 0: for instance a second operand of {0001}{0000}{0000} leaves the first operand unchanged (rather than dividing it by 10^4, as `\__fp_fixed_mul:wwn` would).

```
\cs_new:Npn \__fp_fixed_mul_short:wwn #1#2#3#4#5#6; #7#8#9;
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
      + #1*#7
      \exp_after:wN \__fp_pack:NNNNNw
      \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
        + #1*#8 + #2*#7
        \exp_after:wN \__fp_pack:NNNNNw
        \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
          + #1*#9 + #2*#8 + #3*#7
          \exp_after:wN \__fp_pack:NNNNNw
          \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
            + #2*#9 + #3*#8 + #4*#7
            \exp_after:wN \__fp_pack:NNNNNw
            \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
              + #3*#9 + #4*#8 + #5*#7
              \exp_after:wN \__fp_pack:NNNNNw
              \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
                + #4*#9 + #5*#8 + #6*#7
                + ( #5*#9 + #6*#8 + #6*#9 / \c__fp_myriad_int )
                / \c__fp_myriad_int ; ;
  }
```

(*End of definition for* `\__fp_fixed_mul_short:wwn`.)

75.4 Dividing a fixed point number by a small integer

__fp_fixed_div_int:wwN
__fp_fixed_div_int:wnN
__fp_fixed_div_int_auxi:wnn
__fp_fixed_div_int_auxii:wnn
__fp_fixed_div_int_pack:Nw
__fp_fixed_div_int_after:Nw

`\__fp_fixed_div_int:wwN ⟨a⟩ ; ⟨n⟩ ; ⟨continuation⟩`

Divides the fixed point number ⟨a⟩ by the (small) integer $0 < \langle n\rangle < 10^4$ and feeds the result to the ⟨*continuation*⟩. There is no bound on a_1.

The arguments of the `i` auxiliary are 1: one of the a_i, 2: n, 3: the `ii` or the `iii` auxiliary. It computes a (somewhat tight) lower bound Q_i for the ratio a_i/n.

The `ii` auxiliary receives Q_i, n, and a_i as arguments. It adds Q_i to a surrounding integer expression, and starts a new one with the initial value 9999, which ensures that the result of this expression has 5 digits. The auxiliary also computes $a_i - n \cdot Q_i$, placing the result in front of the 4 digits of a_{i+1}. The resulting $a'_{i+1} = 10^4(a_i - n \cdot Q_i) + a_{i+1}$ serves as the first argument for a new call to the `i` auxiliary.

When the `iii` auxiliary is called, the situation looks like this:

__fp_fixed_div_int_after:Nw ⟨*continuation*⟩
$-1 + Q_1$
__fp_fixed_div_int_pack:Nw $9999 + Q_2$
__fp_fixed_div_int_pack:Nw $9999 + Q_3$
__fp_fixed_div_int_pack:Nw $9999 + Q_4$
__fp_fixed_div_int_pack:Nw $9999 + Q_5$
__fp_fixed_div_int_pack:Nw 9999
__fp_fixed_div_int_auxii:wnn Q_6 ; {⟨n⟩} {⟨a_6⟩}

where expansion is happening from the last line up. The iii auxiliary adds $Q_6 + 2 \simeq a_6/n + 1$ to the last 9999, giving the integer closest to $10000 + a_6/n$.

Each pack auxiliary receives 5 digits followed by a semicolon. The first digit is added as a carry to the integer expression above, and the 4 other digits are braced. Each call to the pack auxiliary thus produces one brace group. The last brace group is produced by the after auxiliary, which places the ⟨*continuation*⟩ as appropriate.

```
\cs_new:Npn \__fp_fixed_div_int:wwN #1#2#3#4#5#6 ; #7 ; #8
  {
    \exp_after:wN \__fp_fixed_div_int_after:Nw
    \exp_after:wN #8
    \int_value:w \__fp_int_eval:w - 1
      \__fp_fixed_div_int:wnN
      #1; {#7} \__fp_fixed_div_int_auxi:wnn
      #2; {#7} \__fp_fixed_div_int_auxi:wnn
      #3; {#7} \__fp_fixed_div_int_auxi:wnn
      #4; {#7} \__fp_fixed_div_int_auxi:wnn
      #5; {#7} \__fp_fixed_div_int_auxi:wnn
      #6; {#7} \__fp_fixed_div_int_auxii:wnn ;
  }
\cs_new:Npn \__fp_fixed_div_int:wnN #1; #2 #3
  {
    \exp_after:wN #3
    \int_value:w \__fp_int_eval:w #1 / #2 - 1 ;
    {#2}
    {#1}
  }
\cs_new:Npn \__fp_fixed_div_int_auxi:wnn #1; #2 #3
  {
    + #1
    \exp_after:wN \__fp_fixed_div_int_pack:Nw
    \int_value:w \__fp_int_eval:w 9999
      \exp_after:wN \__fp_fixed_div_int:wnN
      \int_value:w \__fp_int_eval:w #3 - #1*#2 \__fp_int_eval_end:
  }
\cs_new:Npn \__fp_fixed_div_int_auxii:wnn #1; #2 #3 { + #1 + 2 ; }
\cs_new:Npn \__fp_fixed_div_int_pack:Nw #1 #2; { + #1; {#2} }
\cs_new:Npn \__fp_fixed_div_int_after:Nw #1 #2; { #1 {#2} }
```

(*End of definition for* __fp_fixed_div_int:wwN *and others.*)

75.5 Adding and subtracting fixed points

__fp_fixed_add:wwn
__fp_fixed_sub:wwn
__fp_fixed_add:Nnnnnwnn
__fp_fixed_add:nnNnnnwn
__fp_fixed_add_pack:NNNNNwn
__fp_fixed_add_after:NNNNNwn

__fp_fixed_add:wwn ⟨*a*⟩ ; ⟨*b*⟩ ; {⟨*continuation*⟩}

Computes $a+b$ (resp. $a-b$) and feeds the result to the ⟨*continuation*⟩. This function requires $0 \le a_1, b_1 \le 114748$, its result must be positive (this happens automatically for addition) and its first group must have at most 5 digits: $(a \pm b)_1 < 100000$. The two functions only differ by a sign, hence use a common auxiliary. It would be nice to grab the 12 brace groups in one go; only 9 parameters are allowed. Start by grabbing the sign, $a_1, \ldots, a_4$, the rest of a, and b_1 and b_2. The second auxiliary receives the rest of a, the sign multiplying b, the rest of b, and the ⟨*continuation*⟩ as arguments. After going down through the various level, we go back up, packing digits and bringing the ⟨*continuation*⟩ (#8, then #7) from the end of the argument list to its start.

```
\cs_new:Npn \__fp_fixed_add:wwn { \__fp_fixed_add:Nnnnnwnn + }
\cs_new:Npn \__fp_fixed_sub:wwn { \__fp_fixed_add:Nnnnnwnn - }
\cs_new:Npn \__fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6; #7#8
  {
    \exp_after:wN \__fp_fixed_add_after:NNNNNwn
    \int_value:w \__fp_int_eval:w 9 9999 9998 + #2#3 #1 #7#8
      \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
      \int_value:w \__fp_int_eval:w 1 9999 9998 + #4#5
        \__fp_fixed_add:nnNnnnwn #6 #1
  }
\cs_new:Npn \__fp_fixed_add:nnNnnnwn #1#2 #3 #4#5 #6#7 ; #8
  {
    #3 #4#5
    \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
    \int_value:w \__fp_int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
  }
\cs_new:Npn \__fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6; #7
  { + #1 ; {#7} {#2#3#4#5} {#6} }
\cs_new:Npn \__fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6; #7
  { #7 {#1#2#3#4#5} {#6} }
```

(*End of definition for* `\__fp_fixed_add:wwn` *and others.*)

75.6 Multiplying fixed points

`\__fp_fixed_mul:wwn`
`\__fp_fixed_mul:nnnnnnnw`

`\__fp_fixed_mul:wwn` ⟨*a*⟩ ; ⟨*b*⟩ ; {⟨*continuation*⟩}

Computes $a \times b$ and feeds the result to ⟨*continuation*⟩. This function requires $0 \le a_1, b_1 < 10000$. Once more, we need to play around the limit of 9 arguments for TeX macros. Note that we don't need to obtain an exact rounding, contrarily to the `*` operator, so things could be harder. We wish to perform carries in

$$\begin{aligned} a \times b =& a_1 \cdot b_1 \cdot 10^{-8} \\ &+ (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\ &+ (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1) \cdot 10^{-16} \\ &+ (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\ &+ \Big(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \\ &\qquad + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \\ &\qquad + a_1 \cdot b_5 + a_5 \cdot b_1\Big) \cdot 10^{-24} + O(10^{-24}), \end{aligned}$$

where the $O(10^{-24})$ stands for terms which are at most $5 \cdot 10^{-24}$; ignoring those leads to an error of at most 5 `ulp`. Note how the first 15 terms only depend on $a_1, \ldots, a_4$ and $b_1, \ldots, b_4$, while the last 6 terms only depend on a_1, a_2, a_5, a_6, and the corresponding parts of b. Hence, the first function grabs $a_1, \ldots, a_4$, the rest of a, and $b_1, \ldots, b_4$, and writes the 15 first terms of the expression, including a left parenthesis for the fraction. The `i` auxiliary receives a_5, a_6, b_1, b_2, a_1, a_2, b_5, b_6 and finally the ⟨*continuation*⟩ as arguments. It writes the end of the expression, including the right parenthesis and the denominator of the fraction. The ⟨*continuation*⟩ is finally placed in front of the 6 brace groups by `\__fp_fixed_mul_after:wwn`.

```
\cs_new:Npn \__fp_fixed_mul:wwn #1#2#3#4 #5; #6#7#8#9
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
      \exp_after:wN \__fp_pack:NNNNNw
      \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
        + #1*#6
        \exp_after:wN \__fp_pack:NNNNNw
        \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
          + #1*#7 + #2*#6
          \exp_after:wN \__fp_pack:NNNNNw
          \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
            + #1*#8 + #2*#7 + #3*#6
            \exp_after:wN \__fp_pack:NNNNNw
            \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
              + #1*#9 + #2*#8 + #3*#7 + #4*#6
              \exp_after:wN \__fp_pack:NNNNNw
              \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
                + #2*#9 + #3*#8 + #4*#7
                + ( #3*#9 + #4*#8
                  + \__fp_fixed_mul:nnnnnnnw #5 {#6}{#7}  {#1}{#2}
  }
\cs_new:Npn \__fp_fixed_mul:nnnnnnnw #1#2 #3#4 #5#6 #7#8 ;
  {
    #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c__fp_myriad_int
    + #1*#3 + #5*#7 ; ;
  }
```

(*End of definition for* `\__fp_fixed_mul:wwn` *and* `\__fp_fixed_mul:nnnnnnnw`.)

75.7 Combining product and sum of fixed points

`\__fp_fixed_mul_add:wwwn`
`\__fp_fixed_mul_sub_back:wwwn`
`\__fp_fixed_mul_one_minus_mul:wwn`

`\__fp_fixed_mul_add:wwwn` ⟨*a*⟩ ; ⟨*b*⟩ ; ⟨*c*⟩ ; {⟨*continuation*⟩}
`\__fp_fixed_mul_sub_back:wwwn` ⟨*a*⟩ ; ⟨*b*⟩ ; ⟨*c*⟩ ; {⟨*continuation*⟩}
`\__fp_fixed_one_minus_mul:wwn` ⟨*a*⟩ ; ⟨*b*⟩ ; {⟨*continuation*⟩}

Sometimes called `FMA` (fused multiply-add), these functions compute $a \times b + c$, $c - a \times b$, and $1 - a \times b$ and feed the result to the ⟨*continuation*⟩. Those functions require $0 \leq a_1, b_1, c_1 \leq 10000$. Since those functions are at the heart of the computation of Taylor expansions, we over-optimize them a bit, and in particular we do not factor out the common parts of the three functions.

For definiteness, consider the task of computing $a \times b + c$. We perform carries in

$$\begin{aligned} a \times b + c = &(a_1 \cdot b_1 + c_1 c_2) \cdot 10^{-8} \\ &+ (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\ &+ (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4) \cdot 10^{-16} \\ &+ (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\ &+ \Big(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \\ &\qquad + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \\ &\qquad + a_1 \cdot b_5 + a_5 \cdot b_1 + c_5 c_6 \Big) \cdot 10^{-24} + O(10^{-24}), \end{aligned}$$

where c_1c_2, c_3c_4, c_5c_6 denote the 8-digit number obtained by juxtaposing the two blocks of digits of c, and $\cdot$ denotes multiplication. The task is obviously tough because we have 18 brace groups in front of us.

Each of the three function starts the first two levels (the first, corresponding to 10^{-4}, is empty), with c_1c_2 in the first level, calls the `i` auxiliary with arguments described later, and adds a trailing `+` c_5c_6 `;` `{`⟨*continuation*⟩`}` `;`. The `+` c_5c_6 piece, which is omitted for `\__fp_fixed_one_minus_mul:wwn`, is taken in the integer expression for the 10^{-24} level.

```
\cs_new:Npn \__fp_fixed_mul_add:wwwn #1; #2; #3#4#5#6#7#8;
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \int_value:w \__fp_int_eval:w \c__fp_big_leading_shift_int
      \exp_after:wN \__fp_pack_big:NNNNNNw
      \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int + #3 #4
        \__fp_fixed_mul_add:Nwnnnwnnn +
          + #5 #6 ; #2 ; #1 ; #2 ; +
          + #7 #8 ; ;
  }
\cs_new:Npn \__fp_fixed_mul_sub_back:wwwn #1; #2; #3#4#5#6#7#8;
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \int_value:w \__fp_int_eval:w \c__fp_big_leading_shift_int
      \exp_after:wN \__fp_pack_big:NNNNNNw
      \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int + #3 #4
        \__fp_fixed_mul_add:Nwnnnwnnn -
          + #5 #6 ; #2 ; #1 ; #2 ; -
          + #7 #8 ; ;
  }
\cs_new:Npn \__fp_fixed_one_minus_mul:wwn #1; #2;
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \int_value:w \__fp_int_eval:w \c__fp_big_leading_shift_int
      \exp_after:wN \__fp_pack_big:NNNNNNw
      \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int +
        1 0000 0000
        \__fp_fixed_mul_add:Nwnnnwnnn -
          ; #2 ; #1 ; #2 ; -
          ; ;
  }
```

(*End of definition for* `\__fp_fixed_mul_add:wwwn`*,* `\__fp_fixed_mul_sub_back:wwwn`*, and* `\__fp_fixed_mul_one_minus_mul:wwn`*.*)

__fp_fixed_mul_add:Nwnnnwnnn

`\__fp_fixed_mul_add:Nwnnnwnnn` ⟨*op*⟩ `+` ⟨c_3⟩ ⟨c_4⟩ `;`
⟨*b*⟩ `;` ⟨*a*⟩ `;` ⟨*b*⟩ `;` ⟨*op*⟩
`+` ⟨c_5⟩ ⟨c_6⟩ `;`

Here, ⟨*op*⟩ is either `+` or `-`. Arguments `#3`, `#4`, `#5` are ⟨b_1⟩, ⟨b_2⟩, ⟨b_3⟩; arguments `#7`, `#8`, `#9` are ⟨a_1⟩, ⟨a_2⟩, ⟨a_3⟩. We can build three levels: $a_1 \cdot b_1$ for 10^{-8}, $(a_1 \cdot b_2 + a_2 \cdot b_1)$ for 10^{-12}, and $(a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4)$ for 10^{-16}. The a–b products use the sign `#1`. Note that `#2` is empty for `\__fp_fixed_one_minus_mul:wwn`. We call the `ii` auxiliary for levels 10^{-20} and 10^{-24}, keeping the pieces of ⟨*a*⟩ we've read, but not ⟨*b*⟩, since there is another copy later in the input stream.

```
\cs_new:Npn \__fp_fixed_mul_add:Nwnnnwnnn #1 #2; #3#4#5#6; #7#8#9
  {
    #1 #7*#3
    \exp_after:wN \__fp_pack_big:NNNNNNw
    \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
      #1 #7*#4 #1 #8*#3
      \exp_after:wN \__fp_pack_big:NNNNNNw
      \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
        #1 #7*#5 #1 #8*#4 #1 #9*#3 #2
        \exp_after:wN \__fp_pack_big:NNNNNNw
        \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
          #1 \__fp_fixed_mul_add:nnnnwnnnn {#7}{#8}{#9}
  }
```

(*End of definition for* `\__fp_fixed_mul_add:Nwnnnwnnn`*.*)

__fp_fixed_mul_add:nnnnwnnnn

`\__fp_fixed_mul_add:nnnnwnnnn` ⟨*a*⟩ `;` ⟨*b*⟩ `;` ⟨*op*⟩
`+` ⟨c_5⟩ ⟨c_6⟩ `;`

Level 10^{-20} is $(a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1)$, multiplied by the sign, which was inserted by the `i` auxiliary. Then we prepare level 10^{-24}. We don't have access to all parts of ⟨*a*⟩ and ⟨*b*⟩ needed to make all products. Instead, we prepare the partial expressions

$$b_1 + a_4 \cdot b_2 + a_3 \cdot b_3 + a_2 \cdot b_4 + a_1$$
$$b_2 + a_4 \cdot b_3 + a_3 \cdot b_4 + a_2.$$

Obviously, those expressions make no mathematical sense: we complete them with $a_5 \cdot$ and $\cdot b_5$, and with $a_6 \cdot b_1 + a_5 \cdot$ and $\cdot b_5 + a_1 \cdot b_6$, and of course with the trailing $+ c_5 c_6$. To do all this, we keep a_1, a_5, a_6, and the corresponding pieces of ⟨*b*⟩.

```
\cs_new:Npn \__fp_fixed_mul_add:nnnnwnnnn #1#2#3#4#5; #6#7#8#9
  {
    ( #1*#9 + #2*#8 + #3*#7 + #4*#6 )
    \exp_after:wN \__fp_pack_big:NNNNNNw
    \int_value:w \__fp_int_eval:w \c__fp_big_trailing_shift_int
      \__fp_fixed_mul_add:nnnnwnnwN
        { #6 + #4*#7 + #3*#8 + #2*#9 + #1 }
        { #7 + #4*#8 + #3*#9 + #2 }
        {#1} #5;
        {#6}
  }
```

(*End of definition for* `\__fp_fixed_mul_add:nnnnwnnnn`.)

`\__fp_fixed_mul_add:nnnnwnnwN`

`\__fp_fixed_mul_add:nnnnwnnwN` {⟨*partial*$_1$⟩} {⟨*partial*$_2$⟩}
{⟨a_1⟩} {⟨a_5⟩} {⟨a_6⟩} ; {⟨b_1⟩} {⟨b_5⟩} {⟨b_6⟩} ;
⟨*op*⟩ + ⟨c_5⟩ ⟨c_6⟩ ;

Complete the ⟨*partial*$_1$⟩ and ⟨*partial*$_2$⟩ expressions as explained for the `ii` auxiliary. The second one is divided by 10000: this is the carry from level 10^{-28}. The trailing $+ c_5c_6$ is taken into the expression for level 10^{-24}. Note that the total of level 10^{-24} is in the interval $[-5 \cdot 10^8, 6 \cdot 10^8$ (give or take a couple of 10000), hence adding it to the shift gives a 10-digit number, as expected by the packing auxiliaries. See l3fp-aux for the definition of the shifts and packing auxiliaries.

```
\cs_new:Npn \__fp_fixed_mul_add:nnnnwnnwN #1#2 #3#4#5; #6#7#8; #9
  {
    #9 (#4* #1 *#7)
    #9 (#5*#6+#4* #2 *#7+#3*#8) / \c__fp_myriad_int
  }
```

(*End of definition for* `\__fp_fixed_mul_add:nnnnwnnwN`.)

75.8 Extended-precision floating point numbers

In this section we manipulate floating point numbers with roughly 24 significant figures ("extended-precision" numbers, in short, "ep"), which take the form of an integer exponent, followed by a comma, then six groups of digits, ending with a semicolon. The first group of digit may be any non-negative integer, while other groups of digits have 4 digits. In other words, an extended-precision number is an exponent ending in a comma, then a fixed point number. The corresponding value is $0.\langle\mathit{digits}\rangle \cdot 10^{\langle exponent\rangle}$. This convention differs from floating points.

`\__fp_ep_to_fixed:wwn`
`\__fp_ep_to_fixed_auxi:www`
`\__fp_ep_to_fixed_auxii:nnnnnnnwn`

Converts an extended-precision number with an exponent at most 4 and a first block less than 10^8 to a fixed point number whose first block has 12 digits, hopefully starting with many zeros.

```
\cs_new:Npn \__fp_ep_to_fixed:wwn #1,#2
  {
    \exp_after:wN \__fp_ep_to_fixed_auxi:www
    \int_value:w \__fp_int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
    \exp:w \exp_end_continue_f:w
    \prg_replicate:nn { 4 - \int_max:nn {#1} { -32 } } { 0 } ;
  }
\cs_new:Npn \__fp_ep_to_fixed_auxi:www 1#1; #2; #3#4#5#6#7;
  {
    \__fp_pack_eight:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_ep_to_fixed_auxii:nnnnnnnwn ;
    #2 #1#3#4#5#6#7 0000 !
  }
\cs_new:Npn \__fp_ep_to_fixed_auxii:nnnnnnnwn #1#2#3#4#5#6#7; #8! #9
  { #9 {#1#2}{#3}{#4}{#5}{#6}{#7}; }
```

(*End of definition for* `\__fp_ep_to_fixed:wwn`, `\__fp_ep_to_fixed_auxi:www`, *and* `\__fp_ep_to_fixed_auxii:nnnnnnnwn`.)

__fp_ep_to_ep:wwN
__fp_ep_to_ep_loop:N
__fp_ep_to_ep_end:www
__fp_ep_to_ep_zero:ww

Normalize an extended-precision number. More precisely, leading zeros are removed from the mantissa of the argument, decreasing its exponent as appropriate. Then the digits are packed into 6 groups of 4 (discarding any remaining digit, not rounding). Finally, the continuation `#8` is placed before the resulting exponent–mantissa pair. The input exponent may in fact be given as an integer expression. The `loop` auxiliary grabs a digit: if it is 0, decrement the exponent and continue looping, and otherwise call the `end` auxiliary, which places all digits in the right order (the digit that was not 0, and any remaining digits), followed by some 0, then packs them up neatly in $3 \times 2 = 6$ blocks of four. At the end of the day, remove with `\__fp_use_i:ww` any digit that did not make it in the final mantissa (typically only zeros, unless the original first block has more than 4 digits).

```
\cs_new:Npn \__fp_ep_to_ep:wwN #1,#2#3#4#5#6#7; #8
  {
    \exp_after:wN #8
    \int_value:w \__fp_int_eval:w #1 + 4
      \exp_after:wN \use_i:nn
      \exp_after:wN \__fp_ep_to_ep_loop:N
      \int_value:w \__fp_int_eval:w 1 0000 0000 + #2 \__fp_int_eval_end:
      #3#4#5#6#7 ; ; !
  }
\cs_new:Npn \__fp_ep_to_ep_loop:N #1
  {
    \if_meaning:w 0 #1
      - 1
    \else:
      \__fp_ep_to_ep_end:www #1
    \fi:
    \__fp_ep_to_ep_loop:N
  }
\cs_new:Npn \__fp_ep_to_ep_end:www
    #1 \fi: \__fp_ep_to_ep_loop:N #2; #3!
  {
    \fi:
    \if_meaning:w ; #1
      - 2 * \c__fp_max_exponent_int
      \__fp_ep_to_ep_zero:ww
    \fi:
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_use_i:ww , ;
    #1 #2 0000 0000 0000 0000 0000 0000 ;
  }
\cs_new:Npn \__fp_ep_to_ep_zero:ww \fi: #1; #2; #3;
  { \fi: , {1000}{0000}{0000}{0000}{0000}{0000} ; }
```

(*End of definition for* `\__fp_ep_to_ep:wwN` *and others.*)

__fp_ep_compare:wwww
__fp_ep_compare_aux:wwww

In l3fp-trig we need to compare two extended-precision numbers. This is based on the same function for positive floating point numbers, with an extra test if comparing only 16 decimals is not enough to distinguish the numbers. Note that this function only works if the numbers are normalized so that their first block is in $[1000, 9999]$.

```
\cs_new:Npn \__fp_ep_compare:wwww #1,#2#3#4#5#6#7;
```

```
  { \__fp_ep_compare_aux:wwww {#1}{#2}{#3}{#4}{#5}; #6#7; }
\cs_new:Npn \__fp_ep_compare_aux:wwww #1;#2;#3,#4#5#6#7#8#9;
  {
    \if_case:w
      \__fp_compare_npos:nwnw #1; {#3}{#4}{#5}{#6}{#7}; \exp_stop_f:
            \if_int_compare:w #2 = #8#9 \exp_stop_f:
              0
            \else:
              \if_int_compare:w #2 < #8#9 - \fi: 1
            \fi:
    \or:    1
    \else: -1
    \fi:
  }
```

(*End of definition for* `\__fp_ep_compare:wwww` *and* `\__fp_ep_compare_aux:wwww`.)

`\__fp_ep_mul:wwwwn`
`\__fp_ep_mul_raw:wwwwN`

Multiply two extended-precision numbers: first normalize them to avoid losing too much precision, then multiply the mantissas `#2` and `#4` as fixed point numbers, and sum the exponents `#1` and `#3`. The result's first block is in $[100, 9999]$.

```
\cs_new:Npn \__fp_ep_mul:wwwwn #1,#2; #3,#4;
  {
    \__fp_ep_to_ep:wwN #3,#4;
    \__fp_fixed_continue:wn
    {
      \__fp_ep_to_ep:wwN #1,#2;
      \__fp_ep_mul_raw:wwwwN
    }
    \__fp_fixed_continue:wn
  }
\cs_new:Npn \__fp_ep_mul_raw:wwwwN #1,#2; #3,#4; #5
  {
    \__fp_fixed_mul:wwn #2; #4;
    { \exp_after:wN #5 \int_value:w \__fp_int_eval:w #1 + #3 , }
  }
```

(*End of definition for* `\__fp_ep_mul:wwwwn` *and* `\__fp_ep_mul_raw:wwwwN`.)

75.9 Dividing extended-precision numbers

Divisions of extended-precision numbers are difficult to perform with exact rounding: the technique used in l3fp-basics for 16-digit floating point numbers does not generalize easily to 24-digit numbers. Thankfully, there is no need for exact rounding.

Let us call $\langle n\rangle$ the numerator and $\langle d\rangle$ the denominator. After a simple normalization step, we can assume that $\langle n\rangle \in [0.1, 1)$ and $\langle d\rangle \in [0.1, 1)$, and compute $\langle n\rangle/(10\langle d\rangle) \in (0.01, 1)$. In terms of the 6 blocks of digits $\langle n_1\rangle \cdots \langle n_6\rangle$ and the 6 blocks $\langle d_1\rangle \cdots \langle d_6\rangle$, the condition translates to $\langle n_1\rangle, \langle d_1\rangle \in [1000, 9999]$.

We first find an integer estimate $a \simeq 10^8/\langle d\rangle$ by computing

$$\begin{aligned}
\alpha &= \left[\frac{10^9}{\langle d_1\rangle + 1}\right] \\
\beta &= \left[\frac{10^9}{\langle d_1\rangle}\right] \\
a &= 10^3\alpha + (\beta - \alpha)\cdot\left(10^3 - \left[\frac{\langle d_2\rangle}{10}\right]\right) - 1250,
\end{aligned}$$

where $\left[\frac{\bullet}{\bullet}\right]$ denotes ε-TEX's rounding division, which rounds ties away from zero. The idea is to interpolate between $10^3\alpha$ and $10^3\beta$ with a parameter $\langle d_2\rangle/10^4$, so that when $\langle d_2\rangle = 0$ one gets $a = 10^3\beta - 1250 \simeq 10^{12}/\langle d_1\rangle \simeq 10^8/\langle d\rangle$, while when $\langle d_2\rangle = 9999$ one gets $a = 10^3\alpha - 1250 \simeq 10^{12}/(\langle d_1\rangle + 1) \simeq 10^8/\langle d\rangle$. The shift by 1250 helps to ensure that a is an underestimate of the correct value. We shall prove that

$$1 - 1.755\cdot 10^{-5} < \frac{\langle d\rangle a}{10^8} < 1.$$

We can then compute the inverse of $\langle d\rangle a/10^8 = 1 - \epsilon$ using the relation $1/(1-\epsilon) \simeq (1+\epsilon)(1+\epsilon^2) + \epsilon^4$, which is correct up to a relative error of $\epsilon^5 < 1.6\cdot 10^{-24}$. This allows us to find the desired ratio as

$$\frac{\langle n\rangle}{\langle d\rangle} = \frac{\langle n\rangle a}{10^8}\big((1+\epsilon)(1+\epsilon^2) + \epsilon^4\big).$$

Let us prove the upper bound first (multiplied by 10^{15}). Note that $10^7\langle d\rangle < 10^3\langle d_1\rangle + 10^{-1}(\langle d_2\rangle + 1)$, and that ε-TEX's division $\left[\frac{\langle d_2\rangle}{10}\right]$ underestimates $10^{-1}(\langle d_2\rangle + 1)$ by 0.5 at most, as can be checked for each possible last digit of $\langle d_2\rangle$. Then,

$$\begin{aligned}
10^7\langle d\rangle a &< \left(10^3\langle d_1\rangle + \left[\frac{\langle d_2\rangle}{10}\right] + \frac{1}{2}\right)\left(\left(10^3 - \left[\frac{\langle d_2\rangle}{10}\right]\right)\beta + \left[\frac{\langle d_2\rangle}{10}\right]\alpha - 1250\right) && (1) \\
&< \left(10^3\langle d_1\rangle + \left[\frac{\langle d_2\rangle}{10}\right] + \frac{1}{2}\right) && (2) \\
&\quad \left(\left(10^3 - \left[\frac{\langle d_2\rangle}{10}\right]\right)\left(\frac{10^9}{\langle d_1\rangle} + \frac{1}{2}\right) + \left[\frac{\langle d_2\rangle}{10}\right]\left(\frac{10^9}{\langle d_1\rangle + 1} + \frac{1}{2}\right) - 1250\right) && (3) \\
&< \left(10^3\langle d_1\rangle + \left[\frac{\langle d_2\rangle}{10}\right] + \frac{1}{2}\right)\left(\frac{10^{12}}{\langle d_1\rangle} - \left[\frac{\langle d_2\rangle}{10}\right]\frac{10^9}{\langle d_1\rangle(\langle d_1\rangle + 1)} - 750\right) && (4)
\end{aligned}$$

We recognize a quadratic polynomial in $[\langle d_2\rangle/10]$ with a negative leading coefficient: this polynomial is bounded above, according to $([\langle d_2\rangle/10] + a)(b - c[\langle d_2\rangle/10]) \leq (b + ca)^2/(4c)$. Hence,

$$10^7\langle d\rangle a < \frac{10^{15}}{\langle d_1\rangle(\langle d_1\rangle + 1)}\left(\langle d_1\rangle + \frac{1}{2} + \frac{1}{4}10^{-3} - \frac{3}{8}\cdot 10^{-9}\langle d_1\rangle(\langle d_1\rangle + 1)\right)^2$$

Since $\langle d_1\rangle$ takes integer values within $[1000, 9999]$, it is a simple programming exercise to check that the squared expression is always less than $\langle d_1\rangle(\langle d_1\rangle + 1)$, hence $10^7\langle d\rangle a < 10^{15}$. The upper bound is proven. We also find that $\frac{3}{8}$ can be replaced by slightly smaller numbers, but nothing less than $0.374563\ldots$, and going back through the derivation of the upper bound, we find that 1250 is as small a shift as we can obtain without breaking the bound.

Now, the lower bound. The same computation as for the upper bound implies

$$10^7 \langle d \rangle a > \left(10^3 \langle d_1 \rangle + \left[\frac{\langle d_2 \rangle}{10}\right] - \frac{1}{2}\right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left[\frac{\langle d_2 \rangle}{10}\right] \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 1750\right)$$

This time, we want to find the minimum of this quadratic polynomial. Since the leading coefficient is still negative, the minimum is reached for one of the extreme values $[y/10] = 0$ or $[y/10] = 100$, and we easily check the bound for those values.

We have proven that the algorithm gives us a precise enough answer. Incidentally, the upper bound that we derived tells us that $a < 10^8/\langle d \rangle \leq 10^9$, hence we can compute a safely as a TeX integer, and even add 10^9 to it to ease grabbing of all the digits. The lower bound implies $10^8 - 1755 < a$, which we do not care about.

`\__fp_ep_div:wwwwn` Compute the ratio of two extended-precision numbers. The result is an extended-precision number whose first block lies in the range $[100, 9999]$, and is placed after the ⟨*continuation*⟩ once we are done. First normalize the inputs so that both first block lie in $[1000, 9999]$, then call `\__fp_ep_div_esti:wwwwn` ⟨*denominator*⟩ ⟨*numerator*⟩, responsible for estimating the inverse of the denominator.

```
\cs_new:Npn \__fp_ep_div:wwwwn #1,#2; #3,#4;
  {
    \__fp_ep_to_ep:wwN #1,#2;
    \__fp_fixed_continue:wn
    {
      \__fp_ep_to_ep:wwN #3,#4;
      \__fp_ep_div_esti:wwwwn
    }
  }
```

(*End of definition for* `\__fp_ep_div:wwwwn`.)

`\__fp_ep_div_esti:wwwwn`
`\__fp_ep_div_estii:wwnnwwn`
`\__fp_ep_div_estiii:NNNNNwwwn`

The `esti` function evaluates $\alpha = 10^9/(\langle d_1 \rangle + 1)$, which is used twice in the expression for a, and combines the exponents `#1` and `#4` (with a shift by 1 because we later compute $\langle n \rangle/(10\langle d \rangle)$. Then the `estii` function evaluates $10^9 + a$, and puts the exponent `#2` after the continuation `#7`: from there on we can forget exponents and focus on the mantissa. The `estiii` function multiplies the denominator `#7` by $10^{-8}a$ (obtained as a split into the single digit `#1` and two blocks of 4 digits, `#2#3#4#5` and `#6`). The result $10^{-8}a\langle d \rangle = (1 - \epsilon)$, and a partially packed $10^{-9}a$ (as a block of four digits, and five individual digits, not packed by lack of available macro parameters here) are passed to `\__fp_ep_div_epsi:wnNNNNNn`, which computes $10^{-9}a/(1 - \epsilon)$, that is, $1/(10\langle d \rangle)$ and we finally multiply this by the numerator `#8`.

```
\cs_new:Npn \__fp_ep_div_esti:wwwwn #1,#2#3; #4,
  {
    \exp_after:wN \__fp_ep_div_estii:wwnnwwn
    \int_value:w \__fp_int_eval:w 10 0000 0000 / ( #2 + 1 )
      \exp_after:wN ;
    \int_value:w \__fp_int_eval:w #4 - #1 + 1 ,
    {#2} #3;
  }
\cs_new:Npn \__fp_ep_div_estii:wwnnwwn #1; #2,#3#4#5; #6; #7
  {
```

```
    \exp_after:wN \__fp_ep_div_estiii:NNNNNwwwn
    \int_value:w \__fp_int_eval:w 10 0000 0000 - 1750
      + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
    {#3}{#4}#5; #6; { #7 #2, }
  }
\cs_new:Npn \__fp_ep_div_estiii:NNNNNwwwn 1#1#2#3#4#5#6; #7;
  {
    \__fp_fixed_mul_short:wwn #7; {#1}{#2#3#4#5}{#6};
    \__fp_ep_div_epsi:wnNNNNNn {#1#2#3#4}#5#6
    \__fp_fixed_mul:wwn
  }
```

(*End of definition for* `\__fp_ep_div_esti:wwwwn`*,* `\__fp_ep_div_estii:wwnnwwn`*, and* `\__fp_ep_div_estiii:NNNNNwwwn`*.*)

`\__fp_ep_div_epsi:wnNNNNNn`
`\__fp_ep_div_eps_pack:NNNNNw`
`\__fp_ep_div_epsii:wwnNNNNNn`

The bounds shown above imply that the `epsi` function's first operand is $(1-\epsilon)$ with $\epsilon \in [0, 1.755 \cdot 10^{-5}]$. The `epsi` function computes ϵ as $1-(1-\epsilon)$. Since $\epsilon < 10^{-4}$, its first block vanishes and there is no need to explicitly use `#1` (which is 9999). Then `epsii` evaluates $10^{-9}a/(1-\epsilon)$ as $(1+\epsilon^2)(1+\epsilon)(10^{-9}a\epsilon)+10^{-9}a$. Importantly, we compute $10^{-9}a\epsilon$ before multiplying it with the rest, rather than multiplying by ϵ and then $10^{-9}a$, as this second option loses more precision. Also, the combination of `short_mul` and `div_myriad` is both faster and more precise than a simple `mul`.

```
\cs_new:Npn \__fp_ep_div_epsi:wnNNNNNn #1#2#3#4#5#6;
  {
    \exp_after:wN \__fp_ep_div_epsii:wwnNNNNNn
    \int_value:w \__fp_int_eval:w 1 9998 - #2
      \exp_after:wN \__fp_ep_div_eps_pack:NNNNNw
      \int_value:w \__fp_int_eval:w 1 9999 9998 - #3#4
        \exp_after:wN \__fp_ep_div_eps_pack:NNNNNw
        \int_value:w \__fp_int_eval:w 2 0000 0000 - #5#6 ; ;
  }
\cs_new:Npn \__fp_ep_div_eps_pack:NNNNNw #1#2#3#4#5#6;
  { + #1 ; {#2#3#4#5} {#6} }
\cs_new:Npn \__fp_ep_div_epsii:wwnNNNNNn 1#1; #2; #3#4#5#6#7#8
  {
    \__fp_fixed_mul:wwn {0000}{#1}#2; {0000}{#1}#2;
    \__fp_fixed_add_one:wN
    \__fp_fixed_mul:wwn {10000} {#1} #2 ;
    {
      \__fp_fixed_mul_short:wwn {0000}{#1}#2; {#3}{#4#5#6#7}{#8000};
      \__fp_fixed_div_myriad:wn
      \__fp_fixed_mul:wwn
    }
    \__fp_fixed_add:wwn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000};
  }
```

(*End of definition for* `\__fp_ep_div_epsi:wnNNNNNn`*,* `\__fp_ep_div_eps_pack:NNNNNw`*, and* `\__fp_ep_div_epsii:wwnNNNNNn`*.*)

75.10 Inverse square root of extended precision numbers

The idea here is similar to division. Normalize the input, multiplying by powers of 100 until we have $x \in [0.01, 1)$. Then find an integer approximation $r \in [101, 1003]$ of $10^2/\sqrt{x}$, as the fixed point of iterations of the Newton method: essentially $r \mapsto (r + 10^8/(x_1 r))/2$, starting from a guess that optimizes the number of steps before convergence. In fact, just as there is a slight shift when computing divisions to ensure that some inequalities hold, we replace 10^8 by a slightly larger number which ensures that $r^2 x \geq 10^4$. This also causes $r \in [101, 1003]$. Another correction to the above is that the input is actually normalized to $[0.1, 1)$, and we use either 10^8 or 10^9 in the Newton method, depending on the parity of the exponent. Skipping those technical hurdles, once we have the approximation r, we set $y = 10^{-4} r^2 x$ (or rather, the correct power of 10 to get $y \simeq 1$) and compute $y^{-1/2}$ through another application of Newton's method. This time, the starting value is $z = 1$, each step maps $z \mapsto z(1.5 - 0.5 y z^2)$, and we perform a fixed number of steps. Our final result combines r with $y^{-1/2}$ as $x^{-1/2} = 10^{-2} r y^{-1/2}$.

__fp_ep_isqrt:wwn
__fp_ep_isqrt_aux:wwn
__fp_ep_isqrt_auxii:wwnnnwn

First normalize the input, then check the parity of the exponent `#1`. If it is even, the result's exponent will be $-$`#1`/2, otherwise it will be (`#1` $-$ 1)/2 (except in the case where the input was an exact power of 100). The `auxii` function receives as `#1` the result's exponent just computed, as `#2` the starting value for the iteration giving r (the values 168 and 535 lead to the least number of iterations before convergence, on average), as `#3` and `#4` one empty argument and one `0`, depending on the parity of the original exponent, as `#5` and `#6` the normalized mantissa (`#5` $\in [1000, 9999]$), and as `#7` the continuation. It sets up the iteration giving r: the `esti` function thus receives the initial two guesses `#2` and 0, an approximation `#5` of $10^4 x$ (its first block of digits), and the empty/zero arguments `#3` and `#4`, followed by the mantissa and an altered continuation where we have stored the result's exponent.

```
\cs_new:Npn \__fp_ep_isqrt:wwn #1,#2;
  {
    \__fp_ep_to_ep:wwN #1,#2;
    \__fp_ep_isqrt_auxi:wwn
  }
\cs_new:Npn \__fp_ep_isqrt_auxi:wwn #1,
  {
    \exp_after:wN \__fp_ep_isqrt_auxii:wwnnnwn
    \int_value:w \__fp_int_eval:w
      \int_if_odd:nTF {#1}
        { (1 - #1) / 2 , 535 , { 0 } { } }
        { 1 - #1 / 2 , 168 , { } { 0 } }
  }
\cs_new:Npn \__fp_ep_isqrt_auxii:wwnnnwn #1, #2, #3#4 #5#6; #7
  {
    \__fp_ep_isqrt_esti:wwwnnwn #2, 0, #5, {#3} {#4}
      {#5} #6 ; { #7 #1 , }
  }
```

(*End of definition for* `\__fp_ep_isqrt:wwn`*,* `\__fp_ep_isqrt_aux:wwn`*, and* `\__fp_ep_isqrt_auxii:wwnnnwn`*.*)

__fp_ep_isqrt_esti:wwwnnwn
__fp_ep_isqrt_estii:wwwnnwn
__fp_ep_isqrt_estiii:NNNNNwwwn

If the last two approximations gave the same result, we are done: call the `estii` function to clean up. Otherwise, evaluate $(\langle \mathit{prev} \rangle + 1.005 \cdot 10^{8 \text{ or } 9}/(\langle \mathit{prev} \rangle \cdot x))/2$, as the next approximation: omitting the 1.005 factor, this would be Newton's method. We can check by brute force that if #4 is empty (the original exponent was even), the process computes an integer slightly larger than $100/\sqrt{x}$, while if #4 is 0 (the original exponent was odd), the result is an integer slightly larger than $100/\sqrt{x/10}$. Once we are done, we evaluate $100r^2/2$ or $10r^2/2$ (when the exponent is even or odd, respectively) and feed that to `estiii`. This third auxiliary finds $y_{\mathrm{even}}/2 = 10^{-4}r^2x/2$ or $y_{\mathrm{odd}}/2 = 10^{-5}r^2x/2$ (again, depending on earlier parity). A simple program shows that $y \in [1, 1.0201]$. The number $y/2$ is fed to `\__fp_ep_isqrt_epsi:wN`, which computes $1/\sqrt{y}$, and we finally multiply the result by r.

```
\cs_new:Npn \__fp_ep_isqrt_esti:wwwnnwn #1, #2, #3, #4
  {
    \if_int_compare:w #1 = #2 \exp_stop_f:
      \exp_after:wN \__fp_ep_isqrt_estii:wwwnnwn
    \fi:
    \exp_after:wN \__fp_ep_isqrt_esti:wwwnnwn
    \int_value:w \__fp_int_eval:w
      (#1 + 1 0050 0000 #4 / (#1 * #3)) / 2 ,
    #1, #3, {#4}
  }
\cs_new:Npn \__fp_ep_isqrt_estii:wwwnnwn #1, #2, #3, #4#5
  {
    \exp_after:wN \__fp_ep_isqrt_estiii:NNNNNwwwn
    \int_value:w \__fp_int_eval:w 1000 0000 + #2 * #2 #5 * 5
      \exp_after:wN , \int_value:w \__fp_int_eval:w 10000 + #2 ;
  }
\cs_new:Npn \__fp_ep_isqrt_estiii:NNNNNwwwn 1#1#2#3#4#5#6, 1#7#8; #9;
  {
    \__fp_fixed_mul_short:wwn #9; {#1} {#2#3#4#5} {#600} ;
    \__fp_ep_isqrt_epsi:wN
    \__fp_fixed_mul_short:wwn {#7} {#80} {0000} ;
  }
```

(*End of definition for* `\__fp_ep_isqrt_esti:wwwnnwn`*,* `\__fp_ep_isqrt_estii:wwwnnwn`*, and* `\__fp_ep_isqrt_estiii:NNNNNwwwn`*.*)

`\__fp_ep_isqrt_epsi:wN`
`\__fp_ep_isqrt_epsii:wwN`

Here, we receive a fixed point number $y/2$ with $y \in [1, 1.0201]$. Starting from $z = 1$ we iterate $z \mapsto z(3/2 - z^2y/2)$. In fact, we start from the first iteration $z = 3/2 - y/2$ to avoid useless multiplications. The `epsii` auxiliary receives z as #1 and y as #2.

```
\cs_new:Npn \__fp_ep_isqrt_epsi:wN #1;
  {
    \__fp_fixed_sub:wwn {15000}{0000}{0000}{0000}{0000}{0000}; #1;
    \__fp_ep_isqrt_epsii:wwN #1;
    \__fp_ep_isqrt_epsii:wwN #1;
    \__fp_ep_isqrt_epsii:wwN #1;
  }
\cs_new:Npn \__fp_ep_isqrt_epsii:wwN #1; #2;
  {
    \__fp_fixed_mul:wwn #1; #1;
    \__fp_fixed_mul_sub_back:wwwn #2;
      {15000}{0000}{0000}{0000}{0000}{0000};
    \__fp_fixed_mul:wwn #1;
  }
```

(*End of definition for* `\__fp_ep_isqrt_epsi:wN` *and* `\__fp_ep_isqrt_epsii:wwN`*.*)

75.11 Converting from fixed point to floating point

After computing Taylor series, we wish to convert the result from extended precision (with or without an exponent) to the public floating point format. The functions here should be called within an integer expression for the overall exponent of the floating point.

`\__fp_ep_to_float_o:wwN`
`\__fp_ep_inv_to_float_o:wwN`

An extended-precision number is simply a comma-delimited exponent followed by a fixed point number. Leave the exponent in the current integer expression then convert the fixed point number.

```
\cs_new:Npn \__fp_ep_to_float_o:wwN #1,
  { + \__fp_int_eval:w #1 \__fp_fixed_to_float_o:wN }
\cs_new:Npn \__fp_ep_inv_to_float_o:wwN #1,#2;
  {
    \__fp_ep_div:wwwwn 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1,#2;
    \__fp_ep_to_float_o:wwN
  }
```

(*End of definition for* `\__fp_ep_to_float_o:wwN` *and* `\__fp_ep_inv_to_float_o:wwN`.)

`\__fp_fixed_inv_to_float_o:wN`

Another function which reduces to converting an extended precision number to a float.

```
\cs_new:Npn \__fp_fixed_inv_to_float_o:wN
  { \__fp_ep_inv_to_float_o:wwN 0, }
```

(*End of definition for* `\__fp_fixed_inv_to_float_o:wN`.)

`\__fp_fixed_to_float_rad_o:wN`

Converts the fixed point number `#1` from degrees to radians then to a floating point number. This could perhaps remain in l3fp-trig.

```
\cs_new:Npn \__fp_fixed_to_float_rad_o:wN #1;
  {
    \__fp_fixed_mul:wwn #1; {5729}{5779}{5130}{8232}{0876}{7981};
    { \__fp_ep_to_float_o:wwN 2, }
  }
```

(*End of definition for* `\__fp_fixed_to_float_rad_o:wN`.)

`\__fp_fixed_to_float_o:wN`
`\__fp_fixed_to_float_o:Nw`

`... \__fp_int_eval:w` ⟨*exponent*⟩ `\__fp_fixed_to_float_o:wN {`⟨a_1⟩`} {`⟨a_2⟩`} {`⟨a_3⟩`} {`⟨a_4⟩`} {`⟨a_5⟩`} {`⟨a_6⟩`} ;` ⟨*sign*⟩

yields

⟨*exponent'*⟩ `; {`⟨a'_1⟩`} {`⟨a'_2⟩`} {`⟨a'_3⟩`} {`⟨a'_4⟩`} ;`

And the `to_fixed` version gives six brace groups instead of 4, ensuring that $1000 \leq$ ⟨a'_1⟩ ≤ 9999. At this stage, we know that ⟨a_1⟩ is positive (otherwise, it is sign of an error before), and we assume that it is less than 10^8.[11]

```
\cs_new:Npn \__fp_fixed_to_float_o:Nw #1#2;
  { \__fp_fixed_to_float_o:wN #2; #1 }
\cs_new:Npn \__fp_fixed_to_float_o:wN #1#2#3#4#5#6; #7
  { % for the 8-digit-at-the-start thing
    + \__fp_int_eval:w \c__fp_block_int
    \exp_after:wN \exp_after:wN
    \exp_after:wN \__fp_fixed_to_loop:N
```

[11] Bruno: I must double check this assumption.

```
    \exp_after:wN \use_none:n
    \int_value:w \__fp_int_eval:w
      1 0000 0000 + #1   \exp_after:wN \__fp_use_none_stop_f:n
      \int_value:w   1#2 \exp_after:wN \__fp_use_none_stop_f:n
      \int_value:w 1#3#4 \exp_after:wN \__fp_use_none_stop_f:n
      \int_value:w 1#5#6
    \exp_after:wN ;
    \exp_after:wN ;
  }
\cs_new:Npn \__fp_fixed_to_loop:N #1
  {
    \if_meaning:w 0 #1
      - 1
      \exp_after:wN \__fp_fixed_to_loop:N
    \else:
      \exp_after:wN \__fp_fixed_to_loop_end:w
      \exp_after:wN #1
    \fi:
  }
\cs_new:Npn \__fp_fixed_to_loop_end:w #1 #2 ;
  {
    \if_meaning:w ; #1
      \exp_after:wN \__fp_fixed_to_float_zero:w
    \else:
      \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
      \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
      \exp_after:wN \__fp_fixed_to_float_pack:ww
      \exp_after:wN ;
    \fi:
    #1 #2 0000 0000 0000 0000 ;
  }
\cs_new:Npn \__fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
  {
    - 2 * \c__fp_max_exponent_int ;
    {0000} {0000} {0000} {0000} ;
  }
\cs_new:Npn \__fp_fixed_to_float_pack:ww #1 ; #2#3 ; ;
  {
    \if_int_compare:w #2 > 4 \exp_stop_f:
      \exp_after:wN \__fp_fixed_to_float_round_up:wnnnnw
    \fi:
    ; #1 ;
  }
\cs_new:Npn \__fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \int_value:w \__fp_int_eval:w 1 #1#2
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \int_value:w \__fp_int_eval:w 1 #3#4 + 1 ;
  }
```

(*End of definition for* `\__fp_fixed_to_float_o:wN` *and* `\__fp_fixed_to_float_o:Nw`.)

```
⟨/package⟩
```

Chapter 76

l3fp-expo implementation

```
⟨*package⟩
⟨@@=fp⟩
```

__fp_parse_word_exp:N
__fp_parse_word_ln:N
__fp_parse_word_fact:N

Unary functions.

```
\cs_new:Npn \__fp_parse_word_exp:N
  { \__fp_parse_unary_function:NNN \__fp_exp_o:w ? }
\cs_new:Npn \__fp_parse_word_ln:N
  { \__fp_parse_unary_function:NNN \__fp_ln_o:w ? }
\cs_new:Npn \__fp_parse_word_fact:N
  { \__fp_parse_unary_function:NNN \__fp_fact_o:w ? }
```

(*End of definition for* `\__fp_parse_word_exp:N`*,* `\__fp_parse_word_ln:N`*, and* `\__fp_parse_word_fact:N`*.*)

76.1 Logarithm

76.1.1 Work plan

As for many other functions, we filter out special cases in `\__fp_ln_o:w`. Then `\__fp_ln_npos_o:w` receives a positive normal number, which we write in the form $a \cdot 10^b$ with $a \in [0.1, 1)$.

The rest of this section is actually not in sync with the code. Or is the code not in sync with the section? In the current code, $c \in [1, 10]$ *is such that* $0.7 \leq ac < 1.4$.

We are given a positive normal number, of the form $a \cdot 10^b$ with $a \in [0.1, 1)$. To compute its logarithm, we find a small integer $5 \leq c < 50$ such that $0.91 \leq ac/5 < 1.1$, and use the relation

$$\ln(a \cdot 10^b) = b \cdot \ln(10) - \ln(c/5) + \ln(ac/5).$$

The logarithms $\ln(10)$ and $\ln(c/5)$ are looked up in a table. The last term is computed using the following Taylor series of ln near 1:

$$\ln\left(\frac{ac}{5}\right) = \ln\left(\frac{1+t}{1-t}\right) = 2t\left(1 + t^2\left(\frac{1}{3} + t^2\left(\frac{1}{5} + t^2\left(\frac{1}{7} + t^2\left(\frac{1}{9} + \cdots\right)\right)\right)\right)\right)$$

where $t = 1 - 10/(ac + 5)$. We can now see one reason for the choice of $ac \sim 5$: then $ac + 5 = 10(1 - \epsilon)$ with $-0.05 < \epsilon \leq 0.045$, hence

$$t = \frac{\epsilon}{1 - \epsilon} = \epsilon(1 + \epsilon)(1 + \epsilon^2)(1 + \epsilon^4) \ldots,$$

is not too difficult to compute.

76.1.2 Some constants

\c__fp_ln_i_fixed_tl
\c__fp_ln_ii_fixed_tl
\c__fp_ln_iii_fixed_tl
\c__fp_ln_iv_fixed_tl
\c__fp_ln_vi_fixed_tl
\c__fp_ln_vii_fixed_tl
\c__fp_ln_viii_fixed_tl
\c__fp_ln_ix_fixed_tl
\c__fp_ln_x_fixed_tl

A few values of the logarithm as extended fixed point numbers. Those are needed in the implementation. It turns out that we don't need the value of ln(5).

```
\tl_const:Nn \c__fp_ln_i_fixed_tl   { {0000}{0000}{0000}{0000}{0000}{0000};}
\tl_const:Nn \c__fp_ln_ii_fixed_tl  { {6931}{4718}{0559}{9453}{0941}{7232};}
\tl_const:Nn \c__fp_ln_iii_fixed_tl {{10986}{1228}{8668}{1096}{9139}{5245};}
\tl_const:Nn \c__fp_ln_iv_fixed_tl  {{13862}{9436}{1119}{8906}{1883}{4464};}
\tl_const:Nn \c__fp_ln_vi_fixed_tl  {{17917}{5946}{9228}{0550}{0081}{2477};}
\tl_const:Nn \c__fp_ln_vii_fixed_tl {{19459}{1014}{9055}{3133}{0510}{5353};}
\tl_const:Nn \c__fp_ln_viii_fixed_tl{{20794}{4154}{1679}{8359}{2825}{1696};}
\tl_const:Nn \c__fp_ln_ix_fixed_tl  {{21972}{2457}{7336}{2193}{8279}{0490};}
\tl_const:Nn \c__fp_ln_x_fixed_tl   {{23025}{8509}{2994}{0456}{8401}{7991};}
```

(*End of definition for* `\c__fp_ln_i_fixed_tl` *and others.*)

76.1.3 Sign, exponent, and special numbers

__fp_ln_o:w

The logarithm of negative numbers (including $-\infty$ and -0) raises the "invalid" exception. The logarithm of $+0$ is $-\infty$, raising a division by zero exception. The logarithm of $+\infty$ or a `nan` is itself. Positive normal numbers call `\__fp_ln_npos_o:w`.

```
\cs_new:Npn \__fp_ln_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_meaning:w 2 #3
      \__fp_case_use:nw { \__fp_invalid_operation_o:nw { ln } }
    \fi:
    \if_case:w #2 \exp_stop_f:
      \__fp_case_use:nw
        { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
    \or:
    \else:
      \__fp_case_return_same_o:w
    \fi:
    \__fp_ln_npos_o:w \s__fp \__fp_chk:w #2#3#4;
  }
```

(*End of definition for* `\__fp_ln_o:w`.)

76.1.4 Absolute ln

__fp_ln_npos_o:w

We catch the case of a significand very close to 0.1 or to 1. In all other cases, the final result is at least 10^{-4}, and then an error of $0.5 \cdot 10^{-20}$ is acceptable.

```
\cs_new:Npn \__fp_ln_npos_o:w \s__fp \__fp_chk:w 10#1#2#3;
  { %^^A todo: ln(1) should be "exact zero", not "underflow"
    \exp_after:wN \__fp_sanitize:Nw
```

```
    \int_value:w % for the overall sign
      \if_int_compare:w #1 < \c_one_int
        2
      \else:
        0
      \fi:
      \exp_after:wN \exp_stop_f:
      \int_value:w \__fp_int_eval:w % for the exponent
        \__fp_ln_significand:NNNNnnnN #2#3
        \__fp_ln_exponent:wn {#1}
  }
```

(*End of definition for* `\__fp_ln_npos_o:w`.)

`\__fp_ln_significand:NNNNnnnN`

`\__fp_ln_significand:NNNNnnnN` ⟨X_1⟩ {⟨X_2⟩} {⟨X_3⟩} {⟨X_4⟩} ⟨*continuation*⟩

This function expands to

⟨*continuation*⟩ {⟨Y_1⟩} {⟨Y_2⟩} {⟨Y_3⟩} {⟨Y_4⟩} {⟨Y_5⟩} {⟨Y_6⟩} ;

where $Y = -\ln(X)$ as an extended fixed point.

```
\cs_new:Npn \__fp_ln_significand:NNNNnnnN #1#2#3#4
  {
    \exp_after:wN \__fp_ln_x_ii:wnnnn
    \int_value:w
      \if_case:w #1 \exp_stop_f:
      \or:
        \if_int_compare:w #2 < 4 \exp_stop_f:
          \__fp_int_eval:w 10 - #2
        \else:
          6
        \fi:
      \or: 4
      \or: 3
      \or: 2
      \or: 2
      \or: 2
      \else: 1
      \fi:
    ; { #1 #2 #3 #4 }
  }
```

(*End of definition for* `\__fp_ln_significand:NNNNnnnN`.)

`\__fp_ln_x_ii:wnnnn`

We have thus found $c \in [1, 10]$ such that $0.7 \leq ac < 1.4$ in all cases. Compute $1 + x = 1 + ac \in [1.7, 2.4)$.

```
\cs_new:Npn \__fp_ln_x_ii:wnnnn #1; #2#3#4#5
  {
    \exp_after:wN \__fp_ln_div_after:Nw
    \cs:w c__fp_ln_ \__fp_int_to_roman:w #1 _fixed_tl \exp_after:wN \cs_end:
    \int_value:w
      \exp_after:wN \__fp_ln_x_iv:wnnnnnnnn
      \int_value:w \__fp_int_eval:w
        \exp_after:wN \__fp_ln_x_iii_var:NNNNNw
        \int_value:w \__fp_int_eval:w 9999 9990 + #1*#2#3 +
          \exp_after:wN \__fp_ln_x_iii:NNNNNNw
```

```
            \int_value:w \__fp_int_eval:w 10 0000 0000 + #1*#4#5 ;
      {20000} {0000} {0000} {0000}
    } %^^A todo: reoptimize (a generalization attempt failed).
\cs_new:Npn \__fp_ln_x_iii:NNNNNNw #1#2 #3#4#5#6 #7;
  { #1#2; {#3#4#5#6} {#7} }
\cs_new:Npn \__fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6;
  {
    #1#2#3#4#5 + 1 ;
    {#1#2#3#4#5} {#6}
  }
```

The Taylor series to be used is expressed in terms of $t = (x-1)/(x+1) = 1-2/(x+1)$. We now compute the quotient with extended precision, reusing some code from `\__fp_-/_o:ww`. Note that $1+x$ is known exactly.

To reuse notations from l3fp-basics, we want to compute A/Z with $A = 2$ and $Z = x+1$. In l3fp-basics, we considered the case where both A and Z are arbitrary, in the range $[0.1, 1)$, and we had to monitor the growth of the sequence of remainders A, B, C, etc. to ensure that no overflow occurred during the computation of the next quotient. The main source of risk was our choice to define the quotient as roughly $10^9 \cdot A/10^5 \cdot Z$: then A was bound to be below $2.147\cdots$, and this limit was never far.

In our case, we can simply work with $10^8 \cdot A$ and $10^4 \cdot Z$, because our reason to work with higher powers has gone: we needed the integer $y \simeq 10^5 \cdot Z$ to be at least 10^4, and now, the definition $y \simeq 10^4 \cdot Z$ suffices.

Let us thus define $y = \lfloor 10^4 \cdot Z \rfloor + 1 \in (1.7 \cdot 10^4, 2.4 \cdot 10^4]$, and

$$Q_1 = \left\lfloor \frac{\lfloor 10^8 \cdot A \rfloor}{y} - \frac{1}{2} \right\rfloor .$$

(The 1/2 comes from how ε-TeX rounds.) As for division, it is easy to see that $Q_1 \leq 10^4 A/Z$, *i.e.*, Q_1 is an underestimate.

Exactly as we did for division, we set $B = 10^4 A - Q_1 Z$. Then

$$\begin{aligned}
10^4 B &\leq A_1 A_2 . A_3 A_4 - \left(\frac{A_1 A_2}{y} - \frac{3}{2} \right) 10^4 Z \\
&\leq A_1 A_2 \left(1 - \frac{10^4 Z}{y} \right) + 1 + \frac{3}{2} y \\
&\leq 10^8 \frac{A}{y} + 1 + \frac{3}{2} y
\end{aligned}$$

In the same way, and using $1.7 \cdot 10^4 \leq y \leq 2.4 \cdot 10^4$, and convexity, we get

$$
\begin{aligned}
10^4 A &= 2 \cdot 10^4 \\
10^4 B &\leq 10^8 \frac{A}{y} + 1.6y \leq 4.7 \cdot 10^4 \\
10^4 C &\leq 10^8 \frac{B}{y} + 1.6y \leq 5.8 \cdot 10^4 \\
10^4 D &\leq 10^8 \frac{C}{y} + 1.6y \leq 6.3 \cdot 10^4 \\
10^4 E &\leq 10^8 \frac{D}{y} + 1.6y \leq 6.5 \cdot 10^4 \\
10^4 F &\leq 10^8 \frac{E}{y} + 1.6y \leq 6.6 \cdot 10^4
\end{aligned}
$$

Note that we compute more steps than for division: since t is not the end result, we need to know it with more accuracy (on the other hand, the ending is much simpler, as we don't need an exact rounding for transcendental functions, but just a faithful rounding).

`\__fp_ln_x_iv:wnnnnnnnn` ⟨*1 or 2*⟩ ⟨*8d*⟩ ; {⟨*4d*⟩} {⟨*4d*⟩} ⟨*fixed-tl*⟩

The number is x. Compute y by adding 1 to the five first digits.

```
\cs_new:Npn \__fp_ln_x_iv:wnnnnnnnn #1; #2#3#4#5 #6#7#8#9
  {
    \exp_after:wN \__fp_div_significand_pack:NNN
    \int_value:w \__fp_int_eval:w
    \__fp_ln_div_i:w #1 ;
      #6 #7 ; {#8} {#9}
      {#2} {#3} {#4} {#5}
      { \exp_after:wN \__fp_ln_div_ii:wwn \int_value:w #1 }
      { \exp_after:wN \__fp_ln_div_ii:wwn \int_value:w #1 }
      { \exp_after:wN \__fp_ln_div_ii:wwn \int_value:w #1 }
      { \exp_after:wN \__fp_ln_div_ii:wwn \int_value:w #1 }
      { \exp_after:wN \__fp_ln_div_vi:wwn \int_value:w #1 }
  }
\cs_new:Npn \__fp_ln_div_i:w #1;
  {
    \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
    \int_value:w \__fp_int_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
  }
\cs_new:Npn \__fp_ln_div_ii:wwn #1; #2;#3 % y; B1;B2 <- for k=1
  {
    \exp_after:wN \__fp_div_significand_pack:NNN
    \int_value:w \__fp_int_eval:w
      \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
      \int_value:w \__fp_int_eval:w 999999 + #2 #3 / #1 ; % Q2
      #2 #3 ;
  }
\cs_new:Npn \__fp_ln_div_vi:wwn #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
  {
    \exp_after:wN \__fp_div_significand_pack:NNN
```

```
    \int_value:w \__fp_int_eval:w 1000000 + #2 #3 / #1 ; % Q6
  }
```

We now have essentially

> `\__fp_ln_div_after:Nw` ⟨*fixed tl*⟩
> `\__fp_div_significand_pack:NNN` $10^6 + Q_1$
> `\__fp_div_significand_pack:NNN` $10^6 + Q_2$
> `\__fp_div_significand_pack:NNN` $10^6 + Q_3$
> `\__fp_div_significand_pack:NNN` $10^6 + Q_4$
> `\__fp_div_significand_pack:NNN` $10^6 + Q_5$
> `\__fp_div_significand_pack:NNN` $10^6 + Q_6$ `;`
> ⟨*exponent*⟩ `;` ⟨*continuation*⟩

where ⟨*fixed tl*⟩ holds the logarithm of a number in $[1, 10]$, and ⟨*exponent*⟩ is the exponent. Also, the expansion is done backwards. Then `\__fp_div_significand_pack:NNN` puts things in the correct order to add the Q_i together and put semicolons between each piece. Once those have been expanded, we get

> `\__fp_ln_div_after:Nw` ⟨*fixed-tl*⟩ ⟨*1d*⟩ `;` ⟨*4d*⟩ `;` ⟨*4d*⟩ `;`
> ⟨*4d*⟩ `;` ⟨*4d*⟩ `;` ⟨*4d*⟩ `;` ⟨*4d*⟩ `;` ⟨*exponent*⟩ `;`

Just as with division, we know that the first two digits are `1` and `0` because of bounds on the final result of the division $2/(x + 1)$, which is between roughly 0.8 and 1.2. We then compute $1 - 2/(x + 1)$, after testing whether $2/(x + 1)$ is greater than or smaller than 1.

```
\cs_new:Npn \__fp_ln_div_after:Nw #1#2;
  {
    \if_meaning:w 0 #2
      \exp_after:wN \__fp_ln_t_small:Nw
    \else:
      \exp_after:wN \__fp_ln_t_large:NNw
      \exp_after:wN -
    \fi:
    #1
  }
\cs_new:Npn \__fp_ln_t_small:Nw #1 #2; #3; #4; #5; #6; #7;
  {
    \exp_after:wN \__fp_ln_t_large:NNw
    \exp_after:wN + % <sign>
    \exp_after:wN #1
    \int_value:w \__fp_int_eval:w 9999 - #2 \exp_after:wN ;
    \int_value:w \__fp_int_eval:w 9999 - #3 \exp_after:wN ;
    \int_value:w \__fp_int_eval:w 9999 - #4 \exp_after:wN ;
    \int_value:w \__fp_int_eval:w 9999 - #5 \exp_after:wN ;
    \int_value:w \__fp_int_eval:w 9999 - #6 \exp_after:wN ;
    \int_value:w \__fp_int_eval:w 1 0000 - #7 ;
  }
```

> `\__fp_ln_t_large:NNw` ⟨*sign*⟩ ⟨*fixed tl*⟩
> ⟨t_1⟩`;` ⟨t_2⟩ `;` ⟨t_3⟩`;` ⟨t_4⟩`;` ⟨t_5⟩ `;` ⟨t_6⟩`;`
> ⟨*exponent*⟩ `;` ⟨*continuation*⟩

Compute the square $\mathtt{t}^2$, and keep `t` at the end with its sign. We know that $\mathtt{t} < 0.1765$, so every piece has at most 4 digits. However, since we were not careful in `\__fp_ln_t_small:w`, they can have less than 4 digits.

```
\cs_new:Npn \__fp_ln_t_large:NNw #1 #2 #3; #4; #5; #6; #7; #8;
  {
    \exp_after:wN \__fp_ln_square_t_after:w
    \int_value:w \__fp_int_eval:w 9999 0000 + #3*#3
      \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
      \int_value:w \__fp_int_eval:w 9999 0000 + 2*#3*#4
        \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
        \int_value:w \__fp_int_eval:w 9999 0000 + 2*#3*#5 + #4*#4
          \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
          \int_value:w \__fp_int_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
            \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
            \int_value:w \__fp_int_eval:w
              1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
              + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
              % ; ; ;
    \exp_after:wN \__fp_ln_twice_t_after:w
    \int_value:w \__fp_int_eval:w -1 + 2*#3
      \exp_after:wN \__fp_ln_twice_t_pack:Nw
      \int_value:w \__fp_int_eval:w 9999 + 2*#4
        \exp_after:wN \__fp_ln_twice_t_pack:Nw
        \int_value:w \__fp_int_eval:w 9999 + 2*#5
          \exp_after:wN \__fp_ln_twice_t_pack:Nw
          \int_value:w \__fp_int_eval:w 9999 + 2*#6
            \exp_after:wN \__fp_ln_twice_t_pack:Nw
            \int_value:w \__fp_int_eval:w 9999 + 2*#7
              \exp_after:wN \__fp_ln_twice_t_pack:Nw
              \int_value:w \__fp_int_eval:w 10000 + 2*#8 ; ;
    { \__fp_ln_c:NwNw #1 }
    #2
  }
\cs_new:Npn \__fp_ln_twice_t_pack:Nw #1 #2; { + #1 ; {#2} }
\cs_new:Npn \__fp_ln_twice_t_after:w #1; { ;;; {#1} }
\cs_new:Npn \__fp_ln_square_t_pack:NNNNNw #1 #2#3#4#5 #6;
  { + #1#2#3#4#5 ; {#6} }
\cs_new:Npn \__fp_ln_square_t_after:w 1 0 #1#2#3 #4;
  { \__fp_ln_Taylor:wwNw {0#1#2#3} {#4} }
```

(*End of definition for* `\__fp_ln_x_ii:wnnnn`.)

`\__fp_ln_Taylor:wwNw` Denoting $T = t^2$, we get

```
\__fp_ln_Taylor:wwNw
  {⟨T1⟩} {⟨T2⟩} {⟨T3⟩} {⟨T4⟩} {⟨T5⟩} {⟨T6⟩} ; ;
  {⟨(2t)1⟩} {⟨(2t)2⟩} {⟨(2t)3⟩} {⟨(2t)4⟩} {⟨(2t)5⟩} {⟨(2t)6⟩} ;
  { \__fp_ln_c:NwNw ⟨sign⟩ }
  ⟨fixed tl⟩ ⟨exponent⟩ ; ⟨continuation⟩
```

And we want to compute

$$\ln\left(\frac{1+t}{1-t}\right) = 2t\left(1 + T\left(\frac{1}{3} + T\left(\frac{1}{5} + T\left(\frac{1}{7} + T\left(\frac{1}{9} + \cdots\right)\right)\right)\right)\right)$$

The process looks as follows

```
\loop 5; A;
\div_int 5; 1.0; \add A; \mul T; {\loop \eval 5-2;}
\add 0.2; A; \mul T; {\loop \eval 5-2;}
\mul B; T; {\loop 3;}
\loop 3; C;
```

This uses the routine for dividing a number by a small integer ($< 10^4$).

```
\cs_new:Npn \__fp_ln_Taylor:wwNw
  { \__fp_ln_Taylor_loop:www 21 ; {0000}{0000}{0000}{0000}{0000}{0000} ; }
\cs_new:Npn \__fp_ln_Taylor_loop:www #1; #2; #3;
  {
    \if_int_compare:w #1 = \c_one_int
      \__fp_ln_Taylor_break:w
    \fi:
    \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl #1;
    \__fp_fixed_add:wwn #2;
    \__fp_fixed_mul:wwn #3;
    {
      \exp_after:wN \__fp_ln_Taylor_loop:www
      \int_value:w \__fp_int_eval:w #1 - 2 ;
    }
    #3;
  }
\cs_new:Npn \__fp_ln_Taylor_break:w \fi: #1 \__fp_fixed_add:wwn #2#3; #4 ;;
  {
    \fi:
    \exp_after:wN \__fp_fixed_mul:wwn
    \exp_after:wN { \int_value:w \__fp_int_eval:w 10000 + #2 } #3;
  }
```

(*End of definition for* `\__fp_ln_Taylor:wwNw`.)

`\__fp_ln_c:NwNw`

`\__fp_ln_c:NwNw` ⟨*sign*⟩
`{`⟨r_1⟩`} {`⟨r_2⟩`} {`⟨r_3⟩`} {`⟨r_4⟩`} {`⟨r_5⟩`} {`⟨r_6⟩`} ;`
⟨*fixed tl*⟩ ⟨*exponent*⟩ ; ⟨*continuation*⟩

We are now reduced to finding $\ln(c)$ and ⟨*exponent*⟩ $\ln(10)$ in a table, and adding it to the mixture. The first step is to get $\ln(c) - \ln(x) = -\ln(a)$, then we get `b` $\ln(10)$ and add or subtract.

For now, $\ln(x)$ is given as $\cdot 10^0$. Unless both the exponent is 1 and $c = 1$, we shift to working in units of $\cdot 10^4$, since the final result is at least $\ln(10/7) \simeq 0.35$.

```
\cs_new:Npn \__fp_ln_c:NwNw #1 #2; #3
  {
    \if_meaning:w + #1
      \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_sub:wwn
    \else:
      \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_add:wwn
    \fi:
    #3 #2 ;
  }
```

(*End of definition for* `\__fp_ln_c:NwNw`.)

`\__fp_ln_exponent:wn`

```
\__fp_ln_exponent:wn
  {⟨s1⟩} {⟨s2⟩} {⟨s3⟩} {⟨s4⟩} {⟨s5⟩} {⟨s6⟩} ;
  {⟨exponent⟩}
```

Compute ⟨*exponent*⟩ times ln(10). Apart from the cases where ⟨*exponent*⟩ is 0 or 1, the result is necessarily at least $\ln(10) \simeq 2.3$ in magnitude. We can thus drop the least significant 4 digits. In the case of a very large (positive or negative) exponent, we can (and we need to) drop 4 additional digits, since the result is of order 10^4. Naively, one would think that in both cases we can drop 4 more digits than we do, but that would be slightly too tight for rounding to happen correctly. Besides, we already have addition and subtraction for 24 digits fixed point numbers.

```
\cs_new:Npn \__fp_ln_exponent:wn #1; #2
  {
    \if_case:w #2 \exp_stop_f:
      0 \__fp_case_return:nw { \__fp_fixed_to_float_o:Nw 2 }
    \or:
      \exp_after:wN \__fp_ln_exponent_one:ww \int_value:w
    \else:
      \if_int_compare:w #2 > \c_zero_int
        \exp_after:wN \__fp_ln_exponent_small:NNww
        \exp_after:wN 0
        \exp_after:wN \__fp_fixed_sub:wwn \int_value:w
      \else:
        \exp_after:wN \__fp_ln_exponent_small:NNww
        \exp_after:wN 2
        \exp_after:wN \__fp_fixed_add:wwn \int_value:w -
      \fi:
    \fi:
    #2; #1;
  }
```

Now we painfully write all the cases.[12] No overflow nor underflow can happen, except when computing `ln(1)`.

```
\cs_new:Npn \__fp_ln_exponent_one:ww 1; #1;
  {
    0
    \exp_after:wN \__fp_fixed_sub:wwn \c__fp_ln_x_fixed_tl #1;
    \__fp_fixed_to_float_o:wN 0
  }
```

For small exponents, we just drop one block of digits, and set the exponent of the log to 4 (minus any shift coming from leading zeros in the conversion from fixed point to floating point). Note that here the exponent has been made positive.

```
\cs_new:Npn \__fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
  {
    4
    \exp_after:wN \__fp_fixed_mul:wwn
      \c__fp_ln_x_fixed_tl
      {#3}{0000}{0000}{0000}{0000}{0000} ;
    #2
      {0000}{#4}{#5}{#6}{#7}{#8};
    \__fp_fixed_to_float_o:wN #1
  }
```

[12] Bruno: do rounding.

(*End of definition for* `\__fp_ln_exponent:wn`.)

76.2 Exponential

76.2.1 Sign, exponent, and special numbers

`\__fp_exp_o:w`

```
\cs_new:Npn \__fp_exp_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w #2 \exp_stop_f:
      \__fp_case_return_o:Nw \c_one_fp
    \or:
      \exp_after:wN \__fp_exp_normal_o:w
    \or:
      \if_meaning:w 0 #3
        \exp_after:wN \__fp_case_return_o:Nw
        \exp_after:wN \c_inf_fp
      \else:
        \exp_after:wN \__fp_case_return_o:Nw
        \exp_after:wN \c_zero_fp
      \fi:
    \or:
      \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2#3#4;
  }
```

(*End of definition for* `\__fp_exp_o:w`.)

`\__fp_exp_normal_o:w`
`\__fp_exp_pos_o:Nnwnw`
`\__fp_exp_overflow:NN`

```
\cs_new:Npn \__fp_exp_normal_o:w \s__fp \__fp_chk:w 1#1
  {
    \if_meaning:w 0 #1
      \__fp_exp_pos_o:NNwnw + \__fp_fixed_to_float_o:wN
    \else:
      \__fp_exp_pos_o:NNwnw - \__fp_fixed_inv_to_float_o:wN
    \fi:
  }
\cs_new:Npn \__fp_exp_pos_o:NNwnw #1#2#3 \fi: #4#5;
  {
    \fi:
    \if_int_compare:w #4 > \c__fp_max_exp_exponent_int
      \token_if_eq_charcode:NNTF + #1
        { \__fp_exp_overflow:NN \__fp_overflow:w \c_inf_fp }
        { \__fp_exp_overflow:NN \__fp_underflow:w \c_zero_fp }
      \exp:w
    \else:
      \exp_after:wN \__fp_sanitize:Nw
      \exp_after:wN 0
      \int_value:w #1 \__fp_int_eval:w
        \if_int_compare:w #4 < \c_zero_int
          \exp_after:wN \use_i:nn
        \else:
```

```
          \exp_after:wN \use_ii:nn
        \fi:
        {
          0
          \__fp_decimate:nNnnnn { - #4 }
            \__fp_exp_Taylor:Nnnwn
        }
        {
          \__fp_decimate:nNnnnn { \c__fp_prec_int - #4 }
            \__fp_exp_pos_large:NnnNwn
        }
        #5
        {#4}
        #1 #2 0
        \exp:w
    \fi:
    \exp_after:wN \exp_end:
  }
\cs_new:Npn \__fp_exp_overflow:NN #1#2
  {
    \exp_after:wN \exp_after:wN
    \exp_after:wN #1
    \exp_after:wN #2
  }
```

(*End of definition for* `\__fp_exp_normal_o:w`*,* `\__fp_exp_pos_o:Nnwnw`*, and* `\__fp_exp_overflow:NN`*.*)

`\__fp_exp_Taylor:Nnnwn`
`\__fp_exp_Taylor_loop:www`
`\__fp_exp_Taylor_break:Nww`

This function is called for numbers in the range $[10^{-9}, 10^{-1})$. We compute 10 terms of the Taylor series. The first argument is irrelevant (rounding digit used by some other functions). The next three arguments, at least 16 digits, delimited by a semicolon, form a fixed point number, so we pack it in blocks of 4 digits.

```
\cs_new:Npn \__fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
  {
    #6
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_exp_Taylor_ii:ww
    ; #2#3#4 0000 0000 ;
  }
\cs_new:Npn \__fp_exp_Taylor_ii:ww #1; #2;
  { \__fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s__fp_stop }
\cs_new:Npn \__fp_exp_Taylor_loop:www #1; #2; #3;
  {
    \if_int_compare:w #1 = \c_one_int
      \exp_after:wN \__fp_exp_Taylor_break:Nww
    \fi:
    \__fp_fixed_div_int:wwN #3 ; #1 ;
    \__fp_fixed_add_one:wN
    \__fp_fixed_mul:wwn #2 ;
    {
      \exp_after:wN \__fp_exp_Taylor_loop:www
      \int_value:w \__fp_int_eval:w #1 - 1 ;
      #2 ;
```

```
    }
  }
\cs_new:Npn \__fp_exp_Taylor_break:Nww #1 #2; #3 \s__fp_stop
  { \__fp_fixed_add_one:wN #2 ; }
```

(*End of definition for* `\__fp_exp_Taylor:Nnnwn`*,* `\__fp_exp_Taylor_loop:www`*, and* `\__fp_exp_Taylor_-break:Nww`*.*)

`\c__fp_exp_intarray` The integer array has $6 \times 9 \times 4 = 216$ items encoding the values of $\exp(j \times 10^i)$ for $j = 1, \ldots, 9$ and $i = -1, \ldots, 4$. Each value is expressed as $\simeq 10^p \times 0.m_1 m_2 m_3$ with three 8-digit blocks m_1, m_2, m_3 and an integer exponent p (one more than the scientific exponent), and these are stored in the integer array as four items: p, $10^8 + m_1$, $10^8 + m_2$, $10^8 + m_3$. The various exponentials are stored in increasing order of $j \times 10^i$.

Storing this data in an integer array makes it slightly harder to access (slower, too), but uses 16 bytes of memory per exponential stored, while storing as tokens used around 40 tokens; tokens have an especially large footprint in Unicode-aware engines.

```
\intarray_const_from_clist:Nn \c__fp_exp_intarray
  {
          1 , 1 1105 1709 , 1 1807 5647 , 1 6248 1171 ,
          1 , 1 1221 4027 , 1 5816 0169 , 1 8339 2107 ,
          1 , 1 1349 8588 , 1 0757 6003 , 1 1039 8374 ,
          1 , 1 1491 8246 , 1 9764 1270 , 1 3178 2485 ,
          1 , 1 1648 7212 , 1 7070 0128 , 1 1468 4865 ,
          1 , 1 1822 1188 , 1 0039 0508 , 1 9748 7537 ,
          1 , 1 2013 7527 , 1 0747 0476 , 1 5216 2455 ,
          1 , 1 2225 5409 , 1 2849 2467 , 1 6045 7954 ,
          1 , 1 2459 6031 , 1 1115 6949 , 1 6638 0013 ,
          1 , 1 2718 2818 , 1 2845 9045 , 1 2353 6029 ,
          1 , 1 7389 0560 , 1 9893 0650 , 1 2272 3043 ,
          2 , 1 2008 5536 , 1 9231 8766 , 1 7740 9285 ,
          2 , 1 5459 8150 , 1 0331 4423 , 1 9078 1103 ,
          3 , 1 1484 1315 , 1 9102 5766 , 1 0342 1116 ,
          3 , 1 4034 2879 , 1 3492 7351 , 1 2260 8387 ,
          4 , 1 1096 6331 , 1 5842 8458 , 1 5992 6372 ,
          4 , 1 2980 9579 , 1 8704 1728 , 1 2747 4359 ,
          4 , 1 8103 0839 , 1 2757 5384 , 1 0077 1000 ,
          5 , 1 2202 6465 , 1 7948 0671 , 1 6516 9579 ,
          9 , 1 4851 6519 , 1 5409 7902 , 1 7796 9107 ,
         14 , 1 1068 6474 , 1 5815 2446 , 1 2146 9905 ,
         18 , 1 2353 8526 , 1 6837 0199 , 1 8540 7900 ,
         22 , 1 5184 7055 , 1 2858 7072 , 1 4640 8745 ,
         27 , 1 1142 0073 , 1 8981 5684 , 1 2836 6296 ,
         31 , 1 2515 4386 , 1 7091 9167 , 1 0062 6578 ,
         35 , 1 5540 6223 , 1 8439 3510 , 1 0525 7117 ,
         40 , 1 1220 4032 , 1 9431 7840 , 1 8020 0271 ,
         44 , 1 2688 1171 , 1 4181 6135 , 1 4484 1263 ,
         87 , 1 7225 9737 , 1 6812 5749 , 1 2581 7748 ,
        131 , 1 1942 4263 , 1 9524 1255 , 1 9365 8421 ,
        174 , 1 5221 4696 , 1 8976 4143 , 1 9505 8876 ,
        218 , 1 1403 5922 , 1 1785 2837 , 1 4107 3977 ,
        261 , 1 3773 0203 , 1 0092 9939 , 1 8234 0143 ,
        305 , 1 1014 2320 , 1 5473 5004 , 1 5094 5533 ,
        348 , 1 2726 3745 , 1 7211 2566 , 1 5673 6478 ,
        391 , 1 7328 8142 , 1 2230 7421 , 1 7051 8866 ,
```

```
      435 , 1 1970 0711 , 1 1401 7046 , 1 9938 8888 ,
      869 , 1 3881 1801 , 1 9428 4368 , 1 5764 8232 ,
     1303 , 1 7646 2009 , 1 8905 4704 , 1 8893 1073 ,
     1738 , 1 1506 3559 , 1 7005 0524 , 1 9009 7592 ,
     2172 , 1 2967 6283 , 1 8402 3667 , 1 0689 6630 ,
     2606 , 1 5846 4389 , 1 5650 2114 , 1 7278 5046 ,
     3041 , 1 1151 7900 , 1 5080 6878 , 1 2914 4154 ,
     3475 , 1 2269 1083 , 1 0850 6857 , 1 8724 4002 ,
     3909 , 1 4470 3047 , 1 3316 5442 , 1 6408 6591 ,
     4343 , 1 8806 8182 , 1 2566 2921 , 1 5872 6150 ,
     8686 , 1 7756 0047 , 1 2598 6861 , 1 0458 3204 ,
    13029 , 1 6830 5723 , 1 7791 4884 , 1 1932 7351 ,
    17372 , 1 6015 5609 , 1 3095 3052 , 1 3494 7574 ,
    21715 , 1 5297 7951 , 1 6443 0315 , 1 3251 3576 ,
    26058 , 1 4665 6719 , 1 0099 3379 , 1 5527 2929 ,
    30401 , 1 4108 9724 , 1 3326 3186 , 1 5271 5665 ,
    34744 , 1 3618 6973 , 1 3140 0875 , 1 3856 4102 ,
    39087 , 1 3186 9209 , 1 6113 3900 , 1 6705 9685 ,
  }
```

(*End of definition for* `\c__fp_exp_intarray`.)

`\__fp_exp_pos_large:NnnNwn`
`\__fp_exp_large_after:wwn`
`\__fp_exp_large:NwN`
`\__fp_exp_intarray:w`
`\__fp_exp_intarray_aux:w`

The first two arguments are irrelevant (a rounding digit, and a brace group with 8 zeros). The third argument is the integer part of our number, then we have the decimal part delimited by a semicolon, and finally the exponent, in the range $[0,5]$. Remove leading zeros from the integer part: putting `#4` in there too ensures that an integer part of 0 is also removed. Then read digits one by one, looking up $\exp(\langle \mathit{digit}\rangle \cdot 10^{\langle \mathit{exponent}\rangle})$ in a table, and multiplying that to the current total. The loop is done by `\__fp_exp_large:NwN`, whose `#1` is the ⟨*exponent*⟩, `#2` is the current mantissa, and `#3` is the ⟨*digit*⟩. At the end, `\__fp_exp_large_after:wwn` moves on to the Taylor series, eventually multiplied with the mantissa that we have just computed.

```
\cs_new:Npn \__fp_exp_pos_large:NnnNwn #1#2#3 #4#5; #6
  {
    \exp_after:wN \exp_after:wN \exp_after:wN \__fp_exp_large:NwN
    \exp_after:wN \exp_after:wN \exp_after:wN #6
    \exp_after:wN \c__fp_one_fixed_tl
    \int_value:w #3 #4 \exp_stop_f:
    #5 00000 ;
  }
\cs_new:Npn \__fp_exp_large:NwN #1#2; #3
  {
    \if_case:w #3 ~
      \exp_after:wN \__fp_fixed_continue:wn
    \else:
      \exp_after:wN \__fp_exp_intarray:w
      \int_value:w \__fp_int_eval:w 36 * #1 + 4 * #3 \exp_after:wN ;
    \fi:
    #2;
    {
      \if_meaning:w 0 #1
        \exp_after:wN \__fp_exp_large_after:wwn
      \else:
        \exp_after:wN \__fp_exp_large:NwN
        \int_value:w \__fp_int_eval:w #1 - 1 \exp_after:wN \scan_stop:
```

```
      \fi:
    }
  }
\cs_new:Npn \__fp_exp_intarray:w #1 ;
  {
    +
    \__kernel_intarray_item:Nn \c__fp_exp_intarray
      { \__fp_int_eval:w #1 - 3 \scan_stop: }
    \exp_after:wN \use_i:nnn
    \exp_after:wN \__fp_fixed_mul:wwn
    \int_value:w 0
    \exp_after:wN \__fp_exp_intarray_aux:w
    \int_value:w \__kernel_intarray_item:Nn
                   \c__fp_exp_intarray { \__fp_int_eval:w #1 - 2 }
    \exp_after:wN \__fp_exp_intarray_aux:w
    \int_value:w \__kernel_intarray_item:Nn
                   \c__fp_exp_intarray { \__fp_int_eval:w #1 - 1 }
    \exp_after:wN \__fp_exp_intarray_aux:w
    \int_value:w \__kernel_intarray_item:Nn \c__fp_exp_intarray {#1} ; ;
  }
\cs_new:Npn \__fp_exp_intarray_aux:w 1 #1#2#3#4#5 ; { ; {#1#2#3#4} {#5} }
\cs_new:Npn \__fp_exp_large_after:wwn #1; #2; #3
  {
    \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2; {} #3
    \__fp_fixed_mul:wwn #1;
  }
```

(*End of definition for* `\__fp_exp_pos_large:NnnNwn` *and others.*)

76.3 Power

Raising a number a to a power b leads to many distinct situations.

a^b	$-\infty$	$(-\infty,-0)$	$-$integer	± 0	$+$integer	$(0,\infty)$	$+\infty$	nan
$+\infty$	$+0$	$+0$		$+1$	$+\infty$		$+\infty$	nan
$(1,\infty)$	$+0$	$+\lvert a\rvert^b$		$+1$	$+\lvert a\rvert^b$		$+\infty$	nan
$+1$	$+1$	$+1$		$+1$	$+1$		$+1$	$+1$
$(0,1)$	$+\infty$	$+\lvert a\rvert^b$		$+1$	$+\lvert a\rvert^b$		$+0$	nan
$+0$	$+\infty$	$+\infty$		$+1$	$+0$		$+0$	nan
-0	$+\infty$	nan	$(-1)^b\infty$	$+1$	$(-1)^b 0$	$+0$	$+0$	nan
$(-1,0)$	$+\infty$	nan	$(-1)^b\lvert a\rvert^b$	$+1$	$(-1)^b\lvert a\rvert^b$	nan	$+0$	nan
-1	$+1$	nan	$(-1)^b$	$+1$	$(-1)^b$	nan	$+1$	nan
$(-\infty,-1)$	$+0$	nan	$(-1)^b\lvert a\rvert^b$	$+1$	$(-1)^b\lvert a\rvert^b$	nan	$+\infty$	nan
$-\infty$	$+0$	$+0$	$(-1)^b 0$	$+1$	$(-1)^b\infty$	nan	$+\infty$	nan
nan	nan	nan	nan	$+1$	nan	nan	nan	nan

We distinguished in this table the cases of finite (positive or negative) integer exponents, as $(-1)^b$ is defined in that case. One peculiarity of this operation is that $\mathtt{nan}^0 = 1^{\mathtt{nan}} = 1$, because this relation is obeyed for any number, even $\pm\infty$.

`\__fp_^_o:ww` We cram most of the tests into a single function to save csnames. First treat the case $b = 0$: $a^0 = 1$ for any a, even `nan`. Then test the sign of a.

- If it is positive, and a is a normal number, call `\__fp_pow_normal_o:ww` followed by the two `fp` a and b. For $a = +0$ or $+\inf$, call `\__fp_pow_zero_or_inf:ww` instead, to return either $+0$ or $+\infty$ as appropriate.
- If a is a `nan`, then skip to the next semicolon (which happens to be conveniently the end of b) and return `nan`.
- Finally, if a is negative, compute $\mathtt{a}^b$ (`\__fp_pow_normal_o:ww` which ignores the sign of its first operand), and keep an extra copy of a and b (the second brace group, containing { b a }, is inserted between a and b). Then do some tests to find the final sign of the result if it exists.

```
\cs_new:cpn { __fp_ \iow_char:N \^ _o:ww }
    \s__fp \__fp_chk:w #1#2#3; \s__fp \__fp_chk:w #4#5#6;
  {
    \if_meaning:w 0 #4
      \__fp_case_return_o:Nw \c_one_fp
    \fi:
    \if_case:w #2 \exp_stop_f:
      \exp_after:wN \use_i:nn
    \or:
      \__fp_case_return_o:Nw \c_nan_fp
    \else:
      \exp_after:wN \__fp_pow_neg:www
      \exp:w \exp_end_continue_f:w \exp_after:wN \use:nn
    \fi:
    {
      \if_meaning:w 1 #1
        \exp_after:wN \__fp_pow_normal_o:ww
      \else:
        \exp_after:wN \__fp_pow_zero_or_inf:ww
      \fi:
      \s__fp \__fp_chk:w #1#2#3;
    }
    { \s__fp \__fp_chk:w #4#5#6; \s__fp \__fp_chk:w #1#2#3; }
    \s__fp \__fp_chk:w #4#5#6;
  }
```

(*End of definition for* `\__fp_^_o:ww`.)

`\__fp_pow_zero_or_inf:ww` Raising -0 or $-\infty$ to `nan` yields `nan`. For other powers, the result is $+0$ if 0 is raised to a positive power or ∞ to a negative power, and $+\infty$ otherwise. Thus, if the type of a and the sign of b coincide, the result is 0, since those conveniently take the same possible values, 0 and 2. Otherwise, either $a = \pm\infty$ and $b > 0$ and the result is $+\infty$, or $a = \pm 0$ with $b < 0$ and we have a division by zero unless $b = -\infty$.

```
\cs_new:Npn \__fp_pow_zero_or_inf:ww
    \s__fp \__fp_chk:w #1#2; \s__fp \__fp_chk:w #3#4
  {
    \if_meaning:w 1 #4
      \__fp_case_return_same_o:w
    \fi:
    \if_meaning:w #1 #4
      \__fp_case_return_o:Nw \c_zero_fp
    \fi:
```

```
    \if_meaning:w 2 #1
      \__fp_case_return_o:Nw \c_inf_fp
    \fi:
    \if_meaning:w 2 #3
      \__fp_case_return_o:Nw \c_inf_fp
    \else:
      \__fp_case_use:nw
        {
          \__fp_division_by_zero_o:NNww \c_inf_fp ^
            \s__fp \__fp_chk:w #1 #2 ;
        }
    \fi:
    \s__fp \__fp_chk:w #3#4
  }
```

(*End of definition for* `\__fp_pow_zero_or_inf:ww`.)

`\__fp_pow_normal_o:ww` We have in front of us a, and $b \neq 0$, we know that a is a normal number, and we wish to compute $|a|^b$. If $|a| = 1$, we return 1, unless $a = -1$ and b is `nan`. Indeed, returning 1 at this point would wrongly raise "invalid" when the sign is considered. If $|a| \neq 1$, test the type of b:

0 Impossible, we already filtered $b = \pm 0$.

1 Call `\__fp_pow_npos_o:Nww`.

2 Return $+\infty$ or $+0$ depending on the sign of b and whether the exponent of a is positive or not.

3 Return b.

```
\cs_new:Npn \__fp_pow_normal_o:ww
    \s__fp \__fp_chk:w 1 #1#2#3; \s__fp \__fp_chk:w #4#5
  {
    \if:w 0 \__fp_str_if_eq:nn { #2 #3 } { 1 {1000} {0000} {0000} {0000} }
      \if_int_compare:w #4 #1 = 32 \exp_stop_f:
        \exp_after:wN \__fp_case_return_ii_o:ww
      \fi:
      \__fp_case_return_o:Nww \c_one_fp
    \fi:
    \if_case:w #4 \exp_stop_f:
    \or:
      \exp_after:wN \__fp_pow_npos_o:Nww
      \exp_after:wN #5
    \or:
      \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
      \if_int_compare:w #2 > \c_zero_int
        \exp_after:wN \__fp_case_return_o:Nww
        \exp_after:wN \c_inf_fp
      \else:
        \exp_after:wN \__fp_case_return_o:Nww
        \exp_after:wN \c_zero_fp
      \fi:
    \or:
      \__fp_case_return_ii_o:ww
```

```
    \fi:
    \s__fp \__fp_chk:w 1 #1 {#2} #3 ;
    \s__fp \__fp_chk:w #4 #5
  }
```

(*End of definition for* `\__fp_pow_normal_o:ww`.)

`\__fp_pow_npos_o:Nww`

We now know that $a \neq \pm 1$ is a normal number, and b is a normal number too. We want to compute $|a|^b = (|x| \cdot 10^n)^{y \cdot 10^p} = \exp((\ln|x| + n\ln(10)) \cdot y \cdot 10^p) = \exp(z)$. To compute the exponential accurately, we need to know the digits of z up to the 16-th position. Since the exponential of 10^5 is infinite, we only need at most 21 digits, hence the fixed point result of `\__fp_ln_o:w` is precise enough for our needs. Start an integer expression for the decimal exponent of $e^{|z|}$. If z is negative, negate that decimal exponent, and prepare to take the inverse when converting from the fixed point to the floating point result.

```
\cs_new:Npn \__fp_pow_npos_o:Nww #1 \s__fp \__fp_chk:w 1#2#3
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN 0
    \int_value:w
      \if:w #1 \if_int_compare:w #3 > \c_zero_int 0 \else: 2 \fi:
        \exp_after:wN \__fp_pow_npos_aux:NNnww
        \exp_after:wN +
        \exp_after:wN \__fp_fixed_to_float_o:wN
      \else:
        \exp_after:wN \__fp_pow_npos_aux:NNnww
        \exp_after:wN -
        \exp_after:wN \__fp_fixed_inv_to_float_o:wN
      \fi:
      {#3}
  }
```

(*End of definition for* `\__fp_pow_npos_o:Nww`.)

`\__fp_pow_npos_aux:NNnww`

The first argument is the conversion function from fixed point to float. Then comes an exponent and the 4 brace groups of x, followed by b. Compute $-\ln(x)$.

```
\cs_new:Npn \__fp_pow_npos_aux:NNnww #1#2#3#4#5; \s__fp \__fp_chk:w 1#6#7#8;
  {
    #1
    \__fp_int_eval:w
      \__fp_ln_significand:NNNNnnnN #4#5
      \__fp_pow_exponent:wnN {#3}
      \__fp_fixed_mul:wwn #8 {0000}{0000} ;
      \__fp_pow_B:wwN #7;
      #1 #2 0 % fixed_to_float_o:wN
  }
\cs_new:Npn \__fp_pow_exponent:wnN #1; #2
  {
    \if_int_compare:w #2 > \c_zero_int
      \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % n\ln(10) - (-\ln(x))
      \exp_after:wN +
    \else:
      \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % -(|n|\ln(10) + (-\ln(x)))
      \exp_after:wN -
    \fi:
```

```
    #2; #1;
  }
\cs_new:Npn \__fp_pow_exponent:Nwnnnnnw #1#2; #3#4#5#6#7#8;
  { %^^A todo: use that in ln.
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
      \exp_after:wN \__fp_pack:NNNNNw
      \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
        #1#2*23025 - #1 #3
        \exp_after:wN \__fp_pack:NNNNNw
        \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
          #1 #2*8509 - #1 #4
          \exp_after:wN \__fp_pack:NNNNNw
          \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
            #1 #2*2994 - #1 #5
            \exp_after:wN \__fp_pack:NNNNNw
            \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
              #1 #2*0456 - #1 #6
              \exp_after:wN \__fp_pack:NNNNNw
              \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
                #1 #2*8401 - #1 #7
                #1 ( #2*7991 - #8 ) / 1 0000 ; ;
  }
\cs_new:Npn \__fp_pow_B:wwN #1#2#3#4#5#6; #7;
  {
    \if_int_compare:w #7 < \c_zero_int
      \exp_after:wN \__fp_pow_C_neg:w \int_value:w -
    \else:
      \if_int_compare:w #7 < 22 \exp_stop_f:
        \exp_after:wN \__fp_pow_C_pos:w \int_value:w
      \else:
        \exp_after:wN \__fp_pow_C_overflow:w \int_value:w
      \fi:
    \fi:
    #7 \exp_after:wN ;
    \int_value:w \__fp_int_eval:w 10 0000 + #1 \__fp_int_eval_end:
    #2#3#4#5#6 0000 0000 0000 0000 0000 0000 ; %^^A todo: how many 0?
  }
\cs_new:Npn \__fp_pow_C_overflow:w #1; #2; #3
  {
    + 2 * \c__fp_max_exponent_int
    \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl
  }
\cs_new:Npn \__fp_pow_C_neg:w #1 ; 1
  {
    \exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w
    \prg_replicate:nn {#1} {0}
  }
\cs_new:Npn \__fp_pow_C_pos:w #1; 1
  { \__fp_pow_C_pos_loop:wN #1; }
\cs_new:Npn \__fp_pow_C_pos_loop:wN #1; #2
  {
    \if_meaning:w 0 #1
      \exp_after:wN \__fp_pow_C_pack:w
```

```
      \exp_after:wN #2
    \else:
      \if_meaning:w 0 #2
        \exp_after:wN \__fp_pow_C_pos_loop:wN \int_value:w
      \else:
        \exp_after:wN \__fp_pow_C_overflow:w \int_value:w
      \fi:
      \__fp_int_eval:w #1 - 1 \exp_after:wN ;
    \fi:
  }
\cs_new:Npn \__fp_pow_C_pack:w
  {
    \exp_after:wN \__fp_exp_large:NwN
    \exp_after:wN 5
    \c__fp_one_fixed_tl
  }
```

(*End of definition for* `\__fp_pow_npos_aux:NNnww`*.*)

`\__fp_pow_neg:www`
`\__fp_pow_neg_aux:wNN`

This function is followed by three floating point numbers: $\mathtt{a}^b$, $a \in [-\infty, -0]$, and b. If b is an even integer (case -1), $a^b = \mathtt{a}^b$. If b is an odd integer (case 0), $a^b = -\mathtt{a}^b$, obtained by a call to `\__fp_pow_neg_aux:wNN`. Otherwise, the sign is undefined. This is invalid, unless $\mathtt{a}^b$ turns out to be $+0$ or `nan`, in which case we return that as a^b. In particular, since the underflow detection occurs before `\__fp_pow_neg:www` is called, `(-0.1)**(12345.67)` gives $+0$ rather than complaining that the sign is not defined.

```
\cs_new:Npn \__fp_pow_neg:www \s__fp \__fp_chk:w #1#2; #3; #4;
  {
    \if_case:w \__fp_pow_neg_case:w #4 ;
      \exp_after:wN \__fp_pow_neg_aux:wNN
    \or:
      \if_int_compare:w \__fp_int_eval:w #1 / 2 = \c_one_int
        \__fp_invalid_operation_o:Nww ^ #3; #4;
        \exp:w \exp_end_continue_f:w
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_use_none_until_s:w
      \fi:
    \fi:
    \__fp_exp_after_o:w
    \s__fp \__fp_chk:w #1#2;
  }
\cs_new:Npn \__fp_pow_neg_aux:wNN #1 \s__fp \__fp_chk:w #2#3
  {
    \exp_after:wN \__fp_exp_after_o:w
    \exp_after:wN \s__fp
    \exp_after:wN \__fp_chk:w
    \exp_after:wN #2
    \int_value:w \__fp_int_eval:w 2 - #3 \__fp_int_eval_end:
  }
```

(*End of definition for* `\__fp_pow_neg:www` *and* `\__fp_pow_neg_aux:wNN`*.*)

`\__fp_pow_neg_case:w`
`\__fp_pow_neg_case_aux:nnnnn`
`\__fp_pow_neg_case_aux:Nnnw`

This function expects a floating point number, and determines its "parity". It should be used after `\if_case:w` or in an integer expression. It gives -1 if the number is an even integer, 0 if the number is an odd integer, and 1 otherwise. Zeros and $\pm\infty$ are even (because very large finite floating points are even), while `nan` is a non-integer. The sign of normal numbers is irrelevant to parity. After `\__fp_decimate:nNnnnn` the argument `#1` of `\__fp_pow_neg_case_aux:Nnnw` is a rounding digit, `0` if and only if the number was an integer, and `#3` is the 8 least significant digits of that integer.

```
\cs_new:Npn \__fp_pow_neg_case:w \s__fp \__fp_chk:w #1#2#3;
  {
    \if_case:w #1 \exp_stop_f:
           -1
    \or:   \__fp_pow_neg_case_aux:nnnnn #3
    \or:   -1
    \else: 1
    \fi:
    \exp_stop_f:
  }
\cs_new:Npn \__fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
  {
    \if_int_compare:w #1 > \c__fp_prec_int
      -1
    \else:
      \__fp_decimate:nNnnnn { \c__fp_prec_int - #1 }
        \__fp_pow_neg_case_aux:Nnnw
        {#2} {#3} {#4} {#5}
    \fi:
  }
\cs_new:Npn \__fp_pow_neg_case_aux:Nnnw #1#2#3#4 ;
  {
    \if_meaning:w 0 #1
      \if_int_odd:w #3 \exp_stop_f:
        0
      \else:
        -1
      \fi:
    \else:
      1
    \fi:
  }
```

(*End of definition for* `\__fp_pow_neg_case:w`*,* `\__fp_pow_neg_case_aux:nnnnn`*, and* `\__fp_pow_neg_case_aux:Nnnw`*.*)

76.4 Factorial

`\c__fp_fact_max_arg_int` The maximum integer whose factorial fits in the exponent range is 3248, as $3249! \sim 10^{10000.8}$

```
\int_const:Nn \c__fp_fact_max_arg_int { 3248 }
```

(*End of definition for* `\c__fp_fact_max_arg_int`*.*)

`\__fp_fact_o:w` First detect ± 0 and $+\infty$ and `nan`. Then note that factorial of anything with a negative sign (except -0) is undefined. Then call `\__fp_small_int:wTF` to get an integer as the argument, and start a loop. This is not the most efficient way of computing the factorial, but it works all right. Of course we work with 24 digits instead of 16. It is easy to check that computing factorials with this precision is enough.

```
\cs_new:Npn \__fp_fact_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w #2 \exp_stop_f:
      \__fp_case_return_o:Nw \c_one_fp
    \or:
    \or:
      \if_meaning:w 0 #3
        \exp_after:wN \__fp_case_return_same_o:w
      \fi:
    \or:
      \__fp_case_return_same_o:w
    \fi:
    \if_meaning:w 2 #3
      \__fp_case_use:nw { \__fp_invalid_operation_o:fw { fact } }
    \fi:
    \__fp_fact_pos_o:w
    \s__fp \__fp_chk:w #2 #3 #4 ;
  }
```

(*End of definition for* `\__fp_fact_o:w`.)

`\__fp_fact_pos_o:w`
`\__fp_fact_int_o:w`

Then check the input is an integer, and call `\__fp_facorial_int_o:n` with that `int` as an argument. If it's too big the factorial overflows. Otherwise call `\__fp_sanitize:Nw` with a positive sign marker `0` and an integer expression that will mop up any exponent in the calculation.

```
\cs_new:Npn \__fp_fact_pos_o:w #1;
  {
    \__fp_small_int:wTF #1;
      { \__fp_fact_int_o:n }
      { \__fp_invalid_operation_o:fw { fact } #1; }
  }
\cs_new:Npn \__fp_fact_int_o:n #1
  {
    \if_int_compare:w #1 > \c__fp_fact_max_arg_int
      \__fp_case_return:nw
        {
          \exp_after:wN \exp_after:wN \exp_after:wN \__fp_overflow:w
          \exp_after:wN \c_inf_fp
        }
    \fi:
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN 0
    \int_value:w \__fp_int_eval:w
    \__fp_fact_loop_o:w #1 . 4 , { 1 } { } { } { } { } { } ;
  }
```

(*End of definition for* `\__fp_fact_pos_o:w` *and* `\__fp_fact_int_o:w`.)

`\__fp_fact_loop_o:w`

The loop receives an integer `#1` whose factorial we want to compute, which we progressively decrement, and the result so far as an extended-precision number `#2` in the form ⟨*exponent*⟩`,`⟨*mantissa*⟩`;`. The loop goes in steps of two because we compute `#1*#1-1` as an integer expression (it must fit since `#1` is at most 3248), then multiply with the result so far. We don't need to fill in most of the mantissa with zeros because `\__fp_ep_mul:wwwwn` first normalizes the extended precision number to avoid loss of precision.

When reaching a small enough number simply use a table of factorials less than 10^8. This limit is chosen because the normalization step cannot deal with larger integers.

```
\cs_new:Npn \__fp_fact_loop_o:w #1 . #2 ;
  {
    \if_int_compare:w #1 < 12 \exp_stop_f:
      \__fp_fact_small_o:w #1
    \fi:
    \exp_after:wN \__fp_ep_mul:wwwwn
    \exp_after:wN 4 \exp_after:wN ,
    \exp_after:wN { \int_value:w \__fp_int_eval:w #1 * (#1 - 1) }
    { } { } { } { } { } ;
    #2 ;
    {
      \exp_after:wN \__fp_fact_loop_o:w
      \int_value:w \__fp_int_eval:w #1 - 2 .
    }
  }
\cs_new:Npn \__fp_fact_small_o:w #1 \fi: #2 ; #3 ; #4
  {
    \fi:
    \exp_after:wN \__fp_ep_mul:wwwwn
    \exp_after:wN 4 \exp_after:wN ,
    \exp_after:wN
      {
        \int_value:w
        \if_case:w #1 \exp_stop_f:
        1 \or: 1 \or: 2 \or: 6 \or: 24 \or: 120 \or: 720 \or: 5040
        \or: 40320 \or: 362880 \or: 3628800 \or: 39916800
        \fi:
      } { } { } { } { } { } ;
    #3 ;
    \__fp_ep_to_float_o:wwN 0
  }
```

(*End of definition for* `\__fp_fact_loop_o:w`.)

```
⟨/package⟩
```

Chapter 77

l3fp-trig implementation

```
⟨*package⟩
⟨@@=fp⟩
```

__fp_parse_word_acos:N
__fp_parse_word_acosd:N
__fp_parse_word_acsc:N
__fp_parse_word_acscd:N
__fp_parse_word_asec:N
__fp_parse_word_asecd:N
__fp_parse_word_asin:N
__fp_parse_word_asind:N
__fp_parse_word_cos:N
__fp_parse_word_cosd:N
__fp_parse_word_cot:N
__fp_parse_word_cotd:N
__fp_parse_word_csc:N
__fp_parse_word_cscd:N
__fp_parse_word_sec:N
__fp_parse_word_secd:N
__fp_parse_word_sin:N
__fp_parse_word_sind:N
__fp_parse_word_tan:N
__fp_parse_word_tand:N

Unary functions.

```
\tl_map_inline:nn
  {
    {acos} {acsc} {asec} {asin}
    {cos} {cot} {csc} {sec} {sin} {tan}
  }
  {
    \cs_new:cpe { __fp_parse_word_#1:N }
      {
        \exp_not:N \__fp_parse_unary_function:NNN
        \exp_not:c { __fp_#1_o:w }
        \exp_not:N \use_i:nn
      }
    \cs_new:cpe { __fp_parse_word_#1d:N }
      {
        \exp_not:N \__fp_parse_unary_function:NNN
        \exp_not:c { __fp_#1_o:w }
        \exp_not:N \use_ii:nn
      }
  }
```

(*End of definition for* `\__fp_parse_word_acos:N` *and others.*)

__fp_parse_word_acot:N
__fp_parse_word_acotd:N
__fp_parse_word_atan:N
__fp_parse_word_atand:N

Those functions may receive a variable number of arguments.

```
\cs_new:Npn \__fp_parse_word_acot:N
  { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_i:nn }
\cs_new:Npn \__fp_parse_word_acotd:N
  { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_ii:nn }
\cs_new:Npn \__fp_parse_word_atan:N
  { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_i:nn }
\cs_new:Npn \__fp_parse_word_atand:N
  { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_ii:nn }
```

(*End of definition for* `\__fp_parse_word_acot:N` *and others.*)

77.1 Direct trigonometric functions

The approach for all trigonometric functions (sine, cosine, tangent, cotangent, cosecant, and secant), with arguments given in radians or in degrees, is the same.

- Filter out special cases (± 0, $\pm\inf$ and `nan`).
- Keep the sign for later, and work with the absolute value $|x|$ of the argument.
- Small numbers ($|x| < 1$ in radians, $|x| < 10$ in degrees) are converted to fixed point numbers (and to radians if $|x|$ is in degrees).
- For larger numbers, we need argument reduction. Subtract a multiple of $\pi/2$ (in degrees, 90) to bring the number to the range to $[0, \pi/2)$ (in degrees, $[0, 90)$).
- Reduce further to $[0, \pi/4]$ (in degrees, $[0, 45]$) using $\sin x = \cos(\pi/2 - x)$, and when working in degrees, convert to radians.
- Use the appropriate power series depending on the octant $\lfloor \frac{\mathtt{x}}{\pi/4} \rfloor \mod 8$ (in degrees, the same formula with $\pi/4 \to 45$), the sign, and the function to compute.

77.1.1 Filtering special cases

`\__fp_sin_o:w` This function, and its analogs for `cos`, `csc`, `sec`, `tan`, and `cot` instead of `sin`, are followed either by `\use_i:nn` and a float in radians or by `\use_ii:nn` and a float in degrees. The sine of ± 0 or `nan` is the same float. The sine of $\pm\infty$ raises an invalid operation exception with the appropriate function name. Otherwise, call the `trig` function to perform argument reduction and if necessary convert the reduced argument to radians. Then, `\__fp_sin_series_o:NNwwww` is called to compute the Taylor series: this function receives a sign `#3`, an initial octant of 0, and the function `\__fp_ep_to_float_o:wwN` which converts the result of the series to a floating point directly rather than taking its inverse, since $\sin(x) = \#3 \sin|x|$.

```
\cs_new:Npn \__fp_sin_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_case_return_same_o:w
    \or:   \__fp_case_use:nw
             {
               \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
                 \__fp_ep_to_float_o:wwN #3 0
             }
    \or:   \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { sin } { sind } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3 #4;
  }
```

(*End of definition for* `\__fp_sin_o:w`.)

`\__fp_cos_o:w` The cosine of ± 0 is 1. The cosine of $\pm\infty$ raises an invalid operation exception. The cosine of `nan` is itself. Otherwise, the `trig` function reduces the argument to at most half a right-angle and converts if necessary to radians. We then call the same series as for sine, but using a positive sign 0 regardless of the sign of x, and with an initial octant of 2, because $\cos(x) = +\sin(\pi/2 + |x|)$.

```
\cs_new:Npn \__fp_cos_o:w #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_case_return_o:Nw \c_one_fp
    \or:   \__fp_case_use:nw
             {
               \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
                 \__fp_ep_to_float_o:wwN 0 2
             }
    \or:   \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3;
  }
```

(*End of definition for* `\__fp_cos_o:w`.)

`\__fp_csc_o:w` The cosecant of ± 0 is $\pm\infty$ with the same sign, with a division by zero exception (see `\__fp_cot_zero_o:Nfw` defined below), which requires the function name. The cosecant of $\pm\infty$ raises an invalid operation exception. The cosecant of `nan` is itself. Otherwise, the `trig` function performs the argument reduction, and converts if necessary to radians before calling the same series as for sine, using the sign `#3`, a starting octant of 0, and inverting during the conversion from the fixed point sine to the floating point result, because $\csc(x) = \#3\big(\sin|x|\big)^{-1}$.

```
\cs_new:Npn \__fp_csc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
    \or:   \__fp_case_use:nw
             {
               \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
                 \__fp_ep_inv_to_float_o:wwN #3 0
             }
    \or:   \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { csc } { cscd } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3 #4;
  }
```

(*End of definition for* `\__fp_csc_o:w`.)

`\__fp_sec_o:w` The secant of ± 0 is 1. The secant of $\pm\infty$ raises an invalid operation exception. The secant of `nan` is itself. Otherwise, the `trig` function reduces the argument and turns it to radians before calling the same series as for sine, using a positive sign 0, a starting octant of 2, and inverting upon conversion, because $\sec(x) = +1/\sin(\pi/2 + |x|)$.

```
\cs_new:Npn \__fp_sec_o:w #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_case_return_o:Nw \c_one_fp
```

```
    \or:   \__fp_case_use:nw
             {
               \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
                 \__fp_ep_inv_to_float_o:wwN 0 2
             }
    \or:   \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { sec } { secd } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3;
  }
```

(*End of definition for* `\__fp_sec_o:w`.)

`\__fp_tan_o:w` The tangent of ± 0 or `nan` is the same floating point number. The tangent of $\pm\infty$ raises an invalid operation exception. Once more, the `trig` function does the argument reduction step and conversion to radians before calling `\__fp_tan_series_o:NNwwww`, with a sign `#3` and an initial octant of 1 (this shift is somewhat arbitrary). See `\__fp_cot_o:w` for an explanation of the 0 argument.

```
\cs_new:Npn \__fp_tan_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_case_return_same_o:w
    \or:   \__fp_case_use:nw
             {
               \__fp_trig:NNNNNwn #1
                 \__fp_tan_series_o:NNwwww 0 #3 1
             }
    \or:   \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { tan } { tand } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3 #4;
  }
```

(*End of definition for* `\__fp_tan_o:w`.)

`\__fp_cot_o:w`
`\__fp_cot_zero_o:Nfw`

The cotangent of ± 0 is $\pm\infty$ with the same sign, with a division by zero exception (see `\__fp_cot_zero_o:Nfw`. The cotangent of $\pm\infty$ raises an invalid operation exception. The cotangent of `nan` is itself. We use $\cot x = -\tan(\pi/2 + x)$, and the initial octant for the tangent was chosen to be 1, so the octant here starts at 3. The change in sign is obtained by feeding `\__fp_tan_series_o:NNwwww` two signs rather than just the sign of the argument: the first of those indicates whether we compute tangent or cotangent. Those signs are eventually combined.

```
\cs_new:Npn \__fp_cot_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
    \or:   \__fp_case_use:nw
             {
               \__fp_trig:NNNNNwn #1
                 \__fp_tan_series_o:NNwwww 2 #3 3
             }
    \or:   \__fp_case_use:nw
```

```
            { \__fp_invalid_operation_o:fw { #1 { cot } { cotd } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3 #4;
  }
\cs_new:Npn \__fp_cot_zero_o:Nfw #1#2#3 \fi:
  {
    \fi:
    \token_if_eq_meaning:NNTF 0 #1
      { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_inf_fp }
      { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_minus_inf_fp }
    {#2}
  }
```

(*End of definition for* `\__fp_cot_o:w` *and* `\__fp_cot_zero_o:Nfw`.)

77.1.2 Distinguishing small and large arguments

`\__fp_trig:NNNNNwn` The first argument is `\use_i:nn` if the operand is in radians and `\use_ii:nn` if it is in degrees. Arguments `#2` to `#5` control what trigonometric function we compute, and `#6` to `#8` are pieces of a normal floating point number. Call the `_series` function `#2`, with arguments `#3`, either a conversion function (`\__fp_ep_to_float_o:wN` or `\__fp_ep_inv_to_float_o:wN`) or a sign 0 or 2 when computing tangent or cotangent; `#4`, a sign 0 or 2; the octant, computed in an integer expression starting with `#5` and stopped by a period; and a fixed point number obtained from the floating point number by argument reduction (if necessary) and conversion to radians (if necessary). Any argument reduction adjusts the octant accordingly by leaving a (positive) shift into its integer expression. Let us explain the integer comparison. Two of the four `\exp_after:wN` are expanded, the expansion hits the test, which is true if the float is at least 1 when working in radians, and at least 10 when working in degrees. Then one of the remaining `\exp_after:wN` hits `#1`, which picks the `trig` or `trigd` function in whichever branch of the conditional was taken. The final `\exp_after:wN` closes the conditional. At the end of the day, a number is `large` if it is ≥ 1 in radians or ≥ 10 in degrees, and `small` otherwise. All four `trig`/`trigd` auxiliaries receive the operand as an extended-precision number.

```
\cs_new:Npn \__fp_trig:NNNNNwn #1#2#3#4#5 \s__fp \__fp_chk:w 1#6#7#8;
  {
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN #4
    \int_value:w \__fp_int_eval:w #5
      \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
      \if_int_compare:w #7 > #1 0 1 \exp_stop_f:
        #1 \__fp_trig_large:ww \__fp_trigd_large:ww
      \else:
        #1 \__fp_trig_small:ww \__fp_trigd_small:ww
      \fi:
    #7,#8{0000}{0000};
  }
```

(*End of definition for* `\__fp_trig:NNNNNwn`.)

77.1.3 Small arguments

__fp_trig_small:ww

This receives a small extended-precision number in radians and converts it to a fixed point number. Some trailing digits may be lost in the conversion, so we keep the original floating point number around: when computing sine or tangent (or their inverses), the last step is to multiply by the floating point number (as an extended-precision number) rather than the fixed point number. The period serves to end the integer expression for the octant.

```
\cs_new:Npn \__fp_trig_small:ww #1,#2;
  { \__fp_ep_to_fixed:wwn #1,#2; . #1,#2; }
```

(*End of definition for* `\__fp_trig_small:ww`.)

__fp_trigd_small:ww

Convert the extended-precision number to radians, then call `\__fp_trig_small:ww` to massage it in the form appropriate for the `_series` auxiliary.

```
\cs_new:Npn \__fp_trigd_small:ww #1,#2;
  {
    \__fp_ep_mul_raw:wwwwN
      -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;
    \__fp_trig_small:ww
  }
```

(*End of definition for* `\__fp_trigd_small:ww`.)

77.1.4 Argument reduction in degrees

__fp_trigd_large:ww
__fp_trigd_large_auxi:nnnnwNNNN
__fp_trigd_large_auxii:wNw
__fp_trigd_large_auxiii:www

Note that $25\times 360 = 9000$, so $10^{k+1} \equiv 10^k \pmod{360}$ for $k\geq 3$. When the exponent `#1` is very large, we can thus safely replace it by 22 (or even 19). We turn the floating point number into a fixed point number with two blocks of 8 digits followed by five blocks of 4 digits. The original float is $100\times\langle\mathit{block}_1\rangle\cdots\langle\mathit{block}_3\rangle.\langle\mathit{block}_4\rangle\cdots\langle\mathit{block}_7\rangle$, or is equal to it modulo 360 if the exponent `#1` is very large. The first auxiliary finds $\langle\mathit{block}_1\rangle+\langle\mathit{block}_2\rangle \pmod 9$, a single digit, and prepends it to the 4 digits of $\langle\mathit{block}_3\rangle$. It also unpacks $\langle\mathit{block}_4\rangle$ and grabs the 4 digits of $\langle\mathit{block}_7\rangle$. The second auxiliary grabs the $\langle\mathit{block}_3\rangle$ plus any contribution from the first two blocks as `#1`, the first digit of $\langle\mathit{block}_4\rangle$ (just after the decimal point in hundreds of degrees) as `#2`, and the three other digits as `#3`. It finds the quotient and remainder of `#1#2` modulo 9, adds twice the quotient to the integer expression for the octant, and places the remainder (between 0 and 8) before `#3` to form a new $\langle\mathit{block}_4\rangle$. The resulting fixed point number is $x\in[0,0.9]$. If $x\geq 0.45$, we add 1 to the octant and feed $0.9-x$ with an exponent of 2 (to compensate the fact that we are working in units of hundreds of degrees rather than degrees) to `\__fp_trigd_small:ww`. Otherwise, we feed it x with an exponent of 2. The third auxiliary also discards digits which were not packed into the various $\langle\mathit{blocks}\rangle$. Since the original exponent `#1` is at least 2, those are all 0 and no precision is lost (`#6` and `#7` are four 0 each).

```
\cs_new:Npn \__fp_trigd_large:ww #1, #2#3#4#5#6#7;
  {
    \exp_after:wN \__fp_pack_eight:wNNNNNNNN
    \exp_after:wN \__fp_pack_eight:wNNNNNNNN
    \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
    \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
    \exp_after:wN \__fp_trigd_large_auxi:nnnnwNNNN
    \exp_after:wN ;
```

```
    \exp:w \exp_end_continue_f:w
    \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
    #2#3#4#5#6#7 0000 0000 0000 !
  }
\cs_new:Npn \__fp_trigd_large_auxi:nnnnwNNNN #1#2#3#4#5; #6#7#8#9
  {
    \exp_after:wN \__fp_trigd_large_auxii:wNw
    \int_value:w \__fp_int_eval:w #1 + #2
      - (#1 + #2 - 4) / 9 * 9 \__fp_int_eval_end:
    #3;
    #4; #5{#6#7#8#9};
  }
\cs_new:Npn \__fp_trigd_large_auxii:wNw #1; #2#3;
  {
    + (#1#2 - 4) / 9 * 2
    \exp_after:wN \__fp_trigd_large_auxiii:www
    \int_value:w \__fp_int_eval:w #1#2
      - (#1#2 - 4) / 9 * 9 \__fp_int_eval_end: #3 ;
  }
\cs_new:Npn \__fp_trigd_large_auxiii:www #1; #2; #3!
  {
    \if_int_compare:w #1 < 4500 \exp_stop_f:
      \exp_after:wN \__fp_use_i_until_s:nw
      \exp_after:wN \__fp_fixed_continue:wn
    \else:
      + 1
    \fi:
    \__fp_fixed_sub:wwn {9000}{0000}{0000}{0000}{0000}{0000};
      {#1}#2{0000}{0000};
    { \__fp_trigd_small:ww 2, }
  }
```

(*End of definition for* `\__fp_trigd_large:ww` *and others.*)

77.1.5 Argument reduction in radians

Arguments greater or equal to 1 need to be reduced to a range where we only need a few terms of the Taylor series. We reduce to the range $[0, 2\pi]$ by subtracting multiples of 2π, then to the smaller range $[0, \pi/2]$ by subtracting multiples of $\pi/2$ (keeping track of how many times $\pi/2$ is subtracted), then to $[0, \pi/4]$ by mapping $x \to \pi/2 - x$ if appropriate. When the argument is very large, say, 10^{100}, an equally large multiple of 2π must be subtracted, hence we must work with a very good approximation of 2π in order to get a sensible remainder modulo 2π.

Specifically, we multiply the argument by an approximation of $1/(2\pi)$ with 10048 digits, then discard the integer part of the result, keeping 52 digits of the fractional part. From the fractional part of $x/(2\pi)$ we deduce the octant (quotient of the first three digits by 125). We then multiply by 8 or -8 (the latter when the octant is odd), ignore any integer part (related to the octant), and convert the fractional part to an extended precision number, before multiplying by $\pi/4$ to convert back to a value in radians in $[0, \pi/4]$.

It is possible to prove that given the precision of floating points and their range of exponents, the 52 digits may start at most with 24 zeros. The 5 last digits are affected by carries from computations which are not done, hence we are left with at least $52 - 24 - 5 = 23$ significant digits, enough to round correctly up to $0.6 \cdot \mathrm{ulp}$ in all cases.

\c__fp_trig_intarray This integer array stores blocks of 8 decimals of $10^{-16}/(2\pi)$. Each entry is 10^8 plus an 8 digit number storing 8 decimals. In total we store 10112 decimals of $10^{-16}/(2\pi)$. The number of decimals we really need is the maximum exponent plus the number of digits we later need, 52, plus 12 ($4 - 1$ groups of 4 digits). The memory footprint (1/2 byte per digit) is the same as an earlier method of storing the data as a control sequence name, but the major advantage is that we can unpack specific subsets of the digits without unpacking the 10112 decimals.

```
\intarray_const_from_clist:Nn \c__fp_trig_intarray
  {
    100000000, 100000000, 115915494, 130918953, 135768883, 176337251,
    143620344, 159645740, 145644874, 176673440, 158896797, 163422653,
    150901138, 102766253, 108595607, 128427267, 157958036, 189291184,
    161145786, 152877967, 141073169, 198392292, 139966937, 140907757,
    130777463, 196925307, 168871739, 128962173, 197661693, 136239024,
    117236290, 111832380, 111422269, 197557159, 140461890, 108690267,
    139561204, 189410936, 193784408, 155287230, 199946443, 140024867,
    123477394, 159610898, 132309678, 130749061, 166986462, 180469944,
    186521878, 181574786, 156696424, 110389958, 174139348, 160998386,
    180991999, 162442875, 158517117, 188584311, 117518767, 116054654,
    175369880, 109739460, 136475933, 137680593, 102494496, 163530532,
    171567755, 103220324, 177781639, 171660229, 146748119, 159816584,
    106060168, 103035998, 113391198, 174988327, 186654435, 127975507,
    100162406, 177564388, 184957131, 108801221, 199376147, 168137776,
    147378906, 133068046, 145797848, 117613124, 127314069, 196077502,
    145002977, 159857089, 105690279, 167851315, 125210016, 131774602,
    109248116, 106240561, 145620314, 164840892, 148459191, 143521157,
    154075562, 100871526, 160680221, 171591407, 157474582, 172259774,
    162853998, 175155329, 139081398, 117724093, 158254797, 107332871,
    190406999, 175907657, 170784934, 170393589, 182808717, 134256403,
    166895116, 162545705, 194332763, 112686500, 126122717, 197115321,
    112599504, 138667945, 103762556, 108363171, 116952597, 158128224,
    194162333, 143145106, 112353687, 185631136, 136692167, 114206974,
    169601292, 150578336, 105311960, 185945098, 139556718, 170995474,
    165104316, 123815517, 158083944, 129799709, 199505254, 138756612,
    194458833, 106846050, 178529151, 151410404, 189298850, 163881607,
    176196993, 107341038, 199957869, 118905980, 193737772, 106187543,
    122271893, 101366255, 126123878, 103875388, 181106814, 106765434,
    108282785, 126933426, 179955607, 107903860, 160352738, 199624512,
    159957492, 176297023, 159409558, 143011648, 129641185, 157771240,
    157544494, 157021789, 176979240, 194903272, 194770216, 164960356,
    153181535, 144003840, 168987471, 176915887, 163190966, 150696440,
    147769706, 187683656, 177810477, 197954503, 153395758, 130188183,
    186879377, 166124814, 195305996, 155802190, 183598751, 103512712,
    190432315, 180498719, 168687775, 194656634, 162210342, 104440855,
    149785037, 192738694, 129353661, 193778292, 187359378, 143470323,
    102371458, 137923557, 111863634, 119294601, 183182291, 196416500,
    187830793, 131353497, 179099745, 186492902, 167450609, 189368909,
    145883050, 133703053, 180547312, 132158094, 131976760, 132283131,
    141898097, 149822438, 133517435, 169898475, 101039500, 168388003,
    197867235, 199608024, 100273901, 108749548, 154787923, 156826113,
```

199489032, 168997427, 108349611, 149208289, 103776784, 174303550,
145684560, 183671479, 130845672, 133270354, 185392556, 120208683,
193240995, 162211753, 131839402, 109707935, 170774965, 149880868,
160663609, 168661967, 103747454, 121028312, 119251846, 122483499,
111611495, 166556037, 196967613, 199312829, 196077608, 127799010,
107830360, 102338272, 198790854, 102387615, 157445430, 192601191,
100543379, 198389046, 154921248, 129516070, 172853005, 122721023,
160175233, 113173179, 175931105, 103281551, 109373913, 163964530,
157926071, 180083617, 195487672, 146459804, 173977292, 144810920,
109371257, 186918332, 189588628, 139904358, 168666639, 175673445,
114095036, 137327191, 174311388, 106638307, 125923027, 159734506,
105482127, 178037065, 133778303, 121709877, 134966568, 149080032,
169885067, 141791464, 168350828, 116168533, 114336160, 173099514,
198531198, 119733758, 144420984, 116559541, 152250643, 139431286,
144403838, 183561508, 179771645, 101706470, 167518774, 156059160,
187168578, 157939226, 123475633, 117111329, 198655941, 159689071,
198506887, 144230057, 151919770, 156900382, 118392562, 120338742,
135362568, 108354156, 151729710, 188117217, 195936832, 156488518,
174997487, 108553116, 159830610, 113921445, 144601614, 188452770,
125114110, 170248521, 173974510, 138667364, 103872860, 109967489,
131735618, 112071174, 104788993, 168886556, 192307848, 150230570,
157144063, 163863202, 136852010, 174100574, 185922811, 115721968,
100397824, 175953001, 166958522, 112303464, 118773650, 143546764,
164565659, 171901123, 108476709, 193097085, 191283646, 166919177,
169387914, 133315566, 150669813, 121641521, 100895711, 172862384,
126070678, 145176011, 113450800, 169947684, 122356989, 162488051,
157759809, 153397080, 185475059, 175362656, 149034394, 145420581,
178864356, 183042000, 131509559, 147434392, 152544850, 167491429,
108647514, 142303321, 133245695, 111634945, 167753939, 142403609,
105438335, 152829243, 142203494, 184366151, 146632286, 102477666,
166049531, 140657343, 157553014, 109082798, 180914786, 169343492,
127376026, 134997829, 195701816, 119643212, 133140475, 176289748,
140828911, 174097478, 126378991, 181699939, 148749771, 151989818,
172666294, 160183053, 195832752, 109236350, 168538892, 128468247,
125997252, 183007668, 156937583, 165972291, 198244297, 147406163,
181831139, 158306744, 134851692, 185973832, 137392662, 140243450,
119978099, 140402189, 161348342, 173613676, 144991382, 171541660,
163424829, 136374185, 106122610, 186132119, 198633462, 184709941,
183994274, 129559156, 128333990, 148038211, 175011612, 111667205,
119125793, 103552929, 124113440, 131161341, 112495318, 138592695,
184904438, 146807849, 109739828, 108855297, 104515305, 139914009,
188698840, 188365483, 166522246, 168624087, 125401404, 100911787,
142122045, 123075334, 173972538, 114940388, 141905868, 142311594,
163227443, 139066125, 116239310, 162831953, 123883392, 113153455,
163815117, 152035108, 174595582, 101123754, 135976815, 153401874,
107394340, 136339780, 138817210, 104531691, 182951948, 179591767,
139541778, 179243527, 161740724, 160593916, 102732282, 187946819,
136491289, 149714953, 143255272, 135916592, 198072479, 198580612,
169007332, 118844526, 179433504, 155801952, 149256630, 162048766,
116134365, 133992028, 175452085, 155344144, 109905129, 182727454,
165911813, 122232840, 151166615, 165070983, 175574337, 129548631,
120411217, 116380915, 160616116, 157320000, 183306114, 160618128,
103262586, 195951602, 146321661, 138576614, 180471993, 127077713,
116441201, 159496011, 106328305, 120759583, 148503050, 179095584,

198298218, 167402898, 138551383, 123957020, 180763975, 150429225,
198476470, 171016426, 197438450, 143091658, 164528360, 132493360,
143546572, 137557916, 113663241, 120457809, 196971566, 134022158,
180545794, 131328278, 100552461, 132088901, 187421210, 192448910,
141005215, 149680971, 113720754, 100571096, 134066431, 135745439,
191597694, 135788920, 179342561, 177830222, 137011486, 142492523,
192487287, 113132021, 176673607, 156645598, 127260957, 141566023,
143787436, 129132109, 174858971, 150713073, 191040726, 143541417,
197057222, 165479803, 181512759, 157912400, 125344680, 148220261,
173422990, 101020483, 106246303, 137964746, 178190501, 181183037,
151538028, 179523433, 141955021, 135689770, 191290561, 143178787,
192086205, 174499925, 178975690, 118492103, 124206471, 138519113,
188147564, 102097605, 154895793, 178514140, 141453051, 151583964,
128232654, 106020603, 131189158, 165702720, 186250269, 191639375,
115278873, 160608114, 155694842, 110322407, 177272742, 116513642,
134366992, 171634030, 194053074, 180652685, 109301658, 192136921,
141431293, 171341061, 157153714, 106203978, 147618426, 150297807,
186062669, 169960809, 118422347, 163350477, 146719017, 145045144,
161663828, 146208240, 186735951, 102371302, 190444377, 194085350,
134454426, 133413062, 163074595, 113830310, 122931469, 134466832,
185176632, 182415152, 110179422, 164439571, 181217170, 121756492,
119644493, 196532222, 118765848, 182445119, 109401340, 150443213,
198586286, 121083179, 139396084, 143898019, 114787389, 177233102,
186310131, 148695521, 126205182, 178063494, 157118662, 177825659,
188310053, 151552316, 165984394, 109022180, 163144545, 121212978,
197344714, 188741258, 126822386, 102360271, 109981191, 152056882,
134723983, 158013366, 106837863, 128867928, 161973236, 172536066,
185216856, 132011948, 197807339, 158419190, 166595838, 167852941,
124187182, 117279875, 106103946, 106481958, 157456200, 160892122,
184163943, 173846549, 158993202, 184812364, 133466119, 170732430,
195458590, 173361878, 162906318, 150165106, 126757685, 112163575,
188696307, 145199922, 100107766, 176830946, 198149756, 122682434,
179367131, 108412102, 119520899, 148191244, 140487511, 171059184,
141399078, 189455775, 118462161, 190415309, 134543802, 180893862,
180732375, 178615267, 179711433, 123241969, 185780563, 176301808,
184386640, 160717536, 183213626, 129671224, 126094285, 140110963,
121826276, 151201170, 122552929, 128965559, 146082049, 138409069,
107606920, 103954646, 119164002, 115673360, 117909631, 187289199,
186343410, 186903200, 157966371, 103128612, 135698881, 176403642,
152540837, 109810814, 183519031, 121318624, 172281810, 150845123,
169019064, 166322359, 138872454, 163073727, 128087898, 130041018,
194859136, 173742589, 141812405, 167291912, 138003306, 134499821,
196315803, 186381054, 124578934, 150084553, 128031351, 118843410,
107373060, 159565443, 173624887, 171292628, 198074235, 139074061,
178690578, 144431052, 174262641, 176783005, 182214864, 162289361,
192966929, 192033046, 169332843, 181580535, 164864073, 118444059,
195496893, 153773183, 167266131, 130108623, 158802128, 180432893,
144562140, 147978945, 142337360, 158506327, 104399819, 132635916,
168734194, 136567839, 101281912, 120281622, 195003330, 112236091,
185875592, 101959081, 122415367, 194990954, 148881099, 175891989,
108115811, 163538891, 163394029, 123722049, 184837522, 142362091,
100834097, 156679171, 100841679, 157022331, 178971071, 102928884,
189701309, 195339954, 124415335, 106062584, 139214524, 133864640,
134324406, 157317477, 155340540, 144810061, 177612569, 108474646,

114329765, 143900008, 138265211, 145210162, 136643111, 197987319,
102751191, 144121361, 169620456, 193602633, 161023559, 162140467,
102901215, 167964187, 135746835, 187317233, 110047459, 163339773,
124770449, 118885134, 141536376, 100915375, 164267438, 145016622,
113937193, 106748706, 128815954, 164819775, 119220771, 102367432,
189062690, 170911791, 194127762, 112245117, 123546771, 115640433,
135772061, 166615646, 174474627, 130562291, 133320309, 153340551,
138417181, 194605321, 150142632, 180008795, 151813296, 175497284,
167018836, 157425342, 150169942, 131069156, 134310662, 160434122,
105213831, 158797111, 150754540, 163290657, 102484886, 148697402,
187203725, 198692811, 149360627, 140384233, 128749423, 132178578,
177507355, 171857043, 178737969, 134023369, 102911446, 196144864,
197697194, 134527467, 144296030, 189437192, 154052665, 188907106,
162062575, 150993037, 199766583, 167936112, 181374511, 104971506,
115378374, 135795558, 167972129, 135876446, 130937572, 103221320,
124605656, 161129971, 131027586, 191128460, 143251843, 143269155,
129284585, 173495971, 150425653, 199302112, 118494723, 121323805,
116549802, 190991967, 168151180, 122483192, 151273721, 199792134,
133106764, 121874844, 126215985, 112167639, 167793529, 182985195,
185453921, 106957880, 158685312, 132775454, 133229161, 198905318,
190537253, 191582222, 192325972, 178133427, 181825606, 148823337,
160719681, 101448145, 131983362, 137910767, 112550175, 128826351,
183649210, 135725874, 110356573, 189469487, 154446940, 118175923,
106093708, 128146501, 185742532, 149692127, 164624247, 183221076,
154737505, 168198834, 156410354, 158027261, 125228550, 131543250,
139591848, 191898263, 104987591, 115406321, 103542638, 190012837,
142615518, 178773183, 175862355, 117537850, 169565995, 170028011,
158412588, 170150030, 117025916, 174630208, 142412449, 112839238,
105257725, 114737141, 123102301, 172563968, 130555358, 132628403,
183638157, 168682846, 143304568, 105994018, 170010719, 152092970,
117799058, 132164175, 179868116, 158654714, 177489647, 116547948,
183121404, 131836079, 184431405, 157311793, 149677763, 173989893,
102277656, 107058530, 140837477, 152640947, 143507039, 152145247,
101683884, 107090870, 161471944, 137225650, 128231458, 172995869,
173831689, 171268519, 139042297, 111072135, 107569780, 137262545,
181410950, 138270388, 198736451, 162848201, 180468288, 120582913,
153390138, 135649144, 130040157, 106509887, 192671541, 174507066,
186888783, 143805558, 135011967, 145862340, 180595327, 124727843,
182925939, 157715840, 136885940, 198993925, 152416883, 178793572,
179679516, 154076673, 192703125, 164187609, 162190243, 104699348,
159891990, 160012977, 174692145, 132970421, 167781726, 115178506,
153008552, 155999794, 102099694, 155431545, 127458567, 104403686,
168042864, 184045128, 181182309, 179349696, 127218364, 192935516,
120298724, 169583299, 148193297, 183358034, 159023227, 105261254,
121144370, 184359584, 194433836, 138388317, 175184116, 108817112,
151279233, 137457721, 193398208, 119005406, 132929377, 175306906,
160741530, 149976826, 147124407, 176881724, 186734216, 185881509,
191334220, 175930947, 117385515, 193408089, 157124410, 163472089,
131949128, 180783576, 131158294, 100549708, 191802336, 165960770,
170927599, 101052702, 181508688, 197828549, 143403726, 142729262,
110348701, 139928688, 153550062, 106151434, 130786653, 196085995,
100587149, 139141652, 106530207, 100852656, 124074703, 166073660,
153338052, 163766757, 120188394, 197277047, 122215363, 138511354,
183463624, 161985542, 159938719, 133367482, 104220974, 149956672,

```
    170250544, 164232439, 157506869, 159133019, 137469191, 142980999,
    134242305, 150172665, 121209241, 145596259, 160554427, 159095199,
    168243130, 184279693, 171132070, 121049823, 123819574, 171759855,
    119501864, 163094029, 175943631, 194450091, 191506160, 149228764,
    132319212, 197034460, 193584259, 126727638, 168143633, 109856853,
    127860243, 132141052, 133076065, 188414958, 158718197, 107124299,
    159592267, 181172796, 144388537, 196763139, 127431422, 179531145,
    100064922, 112650013, 132686230, 121550837,
  }
```

(*End of definition for* `\c__fp_trig_intarray`.)

`\__fp_trig_large:ww`
`\__fp_trig_large_auxi:w`
`\__fp_trig_large_auxii:w`
`\__fp_trig_large_auxiii:w`

The exponent `#1` is between 1 and 10000. We wish to look up decimals $10^{\#1-16}/(2\pi)$ starting from the digit $\#1+1$. Since they are stored in batches of 8, compute $\lfloor \#1/8 \rfloor$ and fetch blocks of 8 digits starting there. The numbering of items in `\c__fp_trig_intarray` starts at 1, so the block $\lfloor \#1/8 \rfloor + 1$ contains the digit we want, at one of the eight positions. Each call to `\int_value:w \__kernel_intarray_item:Nn` expands the next, until being stopped by `\__fp_trig_large_auxiii:w` using `\exp_stop_f:`. Once all these blocks are unpacked, the `\exp_stop_f:` and 0 to 7 digits are removed by `\use_none:n...n`. Finally, `\__fp_trig_large_auxii:w` packs 64 digits (there are between 65 and 72 at this point) into groups of 4 and the `auxv` auxiliary is called.

```
\cs_new:Npn \__fp_trig_large:ww #1, #2#3#4#5#6;
  {
    \exp_after:wN \__fp_trig_large_auxi:w
    \int_value:w \__fp_int_eval:w (#1 - 4) / 8 \exp_after:wN ,
    \int_value:w #1 , ;
    {#2}{#3}{#4}{#5} ;
  }
\cs_new:Npn \__fp_trig_large_auxi:w #1, #2,
  {
    \exp_after:wN \exp_after:wN
    \exp_after:wN \__fp_trig_large_auxii:w
    \cs:w
      use_none:n \prg_replicate:nn { #2 - #1 * 8 } { n }
      \exp_after:wN
    \cs_end:
    \int_value:w
    \__kernel_intarray_item:Nn \c__fp_trig_intarray
      { \__fp_int_eval:w #1 + 1 \scan_stop: }
    \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
    \__kernel_intarray_item:Nn \c__fp_trig_intarray
      { \__fp_int_eval:w #1 + 2 \scan_stop: }
    \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
    \__kernel_intarray_item:Nn \c__fp_trig_intarray
      { \__fp_int_eval:w #1 + 3 \scan_stop: }
    \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
    \__kernel_intarray_item:Nn \c__fp_trig_intarray
      { \__fp_int_eval:w #1 + 4 \scan_stop: }
    \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
    \__kernel_intarray_item:Nn \c__fp_trig_intarray
      { \__fp_int_eval:w #1 + 5 \scan_stop: }
    \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
    \__kernel_intarray_item:Nn \c__fp_trig_intarray
      { \__fp_int_eval:w #1 + 6 \scan_stop: }
```

```
    \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
    \__kernel_intarray_item:Nn \c__fp_trig_intarray
      { \__fp_int_eval:w #1 + 7 \scan_stop: }
    \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
    \__kernel_intarray_item:Nn \c__fp_trig_intarray
      { \__fp_int_eval:w #1 + 8 \scan_stop: }
    \exp_after:wN \__fp_trig_large_auxiii:w \int_value:w
    \__kernel_intarray_item:Nn \c__fp_trig_intarray
      { \__fp_int_eval:w #1 + 9 \scan_stop: }
    \exp_stop_f:
  }
\cs_new:Npn \__fp_trig_large_auxii:w
  {
    \__fp_pack_twice_four:wNNNNNNNN \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN \__fp_pack_twice_four:wNNNNNNNN
    \__fp_trig_large_auxv:www ;
  }
\cs_new:Npn \__fp_trig_large_auxiii:w 1 { \exp_stop_f: }
```

(*End of definition for* `\__fp_trig_large:ww` *and others.*)

`\__fp_trig_large_auxv:www`
`\__fp_trig_large_auxvi:wnnnnnnnn`
`\__fp_trig_large_pack:NNNNNw`

First come the first 64 digits of the fractional part of $10^{\#1-16}/(2\pi)$, arranged in 16 blocks of 4, and ending with a semicolon. Then a few more digits of the same fractional part, ending with a semicolon, then 4 blocks of 4 digits holding the significand of the original argument. Multiply the 16-digit significand with the 64-digit fractional part: the `auxvi` auxiliary receives the significand as `#2#3#4#5` and 16 digits of the fractional part as `#6#7#8#9`, and computes one step of the usual ladder of `pack` functions we use for multiplication (see *e.g.,* `\__fp_fixed_mul:wwn`), then discards one block of the fractional part to set things up for the next step of the ladder. We perform 13 such steps, replacing the last `middle` shift by the appropriate `trailing` shift, then discard the significand and remaining 3 blocks from the fractional part, as there are not enough digits to compute any more step in the ladder. The last semicolon closes the ladder, and we return control to the `auxvii` auxiliary.

```
\cs_new:Npn \__fp_trig_large_auxv:www #1; #2; #3;
  {
    \exp_after:wN \__fp_use_i_until_s:nw
    \exp_after:wN \__fp_trig_large_auxvii:w
    \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
      \prg_replicate:nn { 13 }
        { \__fp_trig_large_auxvi:wnnnnnnnn }
      + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
      \__fp_use_i_until_s:nw
      ; #3 #1 ; ;
  }
\cs_new:Npn \__fp_trig_large_auxvi:wnnnnnnnn #1; #2#3#4#5#6#7#8#9
  {
    \exp_after:wN \__fp_trig_large_pack:NNNNNw
    \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
      + #2*#9 + #3*#8 + #4*#7 + #5*#6
      #1; {#2}{#3}{#4}{#5} {#7}{#8}{#9}
  }
```

```
\cs_new:Npn \__fp_trig_large_pack:NNNNNw #1#2#3#4#5#6;
  { + #1#2#3#4#5 ; #6 }
```

(*End of definition for* `\__fp_trig_large_auxv:www`*,* `\__fp_trig_large_auxvi:wnnnnnnnn`*, and* `\__fp_trig_large_pack:NNNNNw`*.*)

`\__fp_trig_large_auxvii:w`
`\__fp_trig_large_auxviii:w`
`\__fp_trig_large_auxix:Nw`
`\__fp_trig_large_auxx:wNNNNN`
`\__fp_trig_large_auxxi:w`

The `auxvii` auxiliary is followed by 52 digits and a semicolon. We find the octant as the integer part of 8 times what follows, or equivalently as the integer part of `#1#2#3`/125, and add it to the surrounding integer expression for the octant. We then compute 8 times the 52-digit number, with a minus sign if the octant is odd. Again, the last `middle` shift is converted to a `trailing` shift. Any integer part (including negative values which come up when the octant is odd) is discarded by `\__fp_use_i_until_s:nw`. The resulting fractional part should then be converted to radians by multiplying by $2\pi/8$, but first, build an extended precision number by abusing `\__fp_ep_to_ep_loop:N` with the appropriate trailing markers. Finally, `\__fp_trig_small:ww` sets up the argument for the functions which compute the Taylor series.

```
\cs_new:Npn \__fp_trig_large_auxvii:w #1#2#3
  {
    \exp_after:wN \__fp_trig_large_auxviii:ww
    \int_value:w \__fp_int_eval:w (#1#2#3 - 62) / 125 ;
    #1#2#3
  }
\cs_new:Npn \__fp_trig_large_auxviii:ww #1;
  {
    + #1
    \if_int_odd:w #1 \exp_stop_f:
      \exp_after:wN \__fp_trig_large_auxix:Nw
      \exp_after:wN -
    \else:
      \exp_after:wN \__fp_trig_large_auxix:Nw
      \exp_after:wN +
    \fi:
  }
\cs_new:Npn \__fp_trig_large_auxix:Nw
  {
    \exp_after:wN \__fp_use_i_until_s:nw
    \exp_after:wN \__fp_trig_large_auxxi:w
    \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
      \prg_replicate:nn { 13 }
        { \__fp_trig_large_auxx:wNNNNN }
      + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
      ;
  }
\cs_new:Npn \__fp_trig_large_auxx:wNNNNN #1; #2 #3#4#5#6
  {
    \exp_after:wN \__fp_trig_large_pack:NNNNNw
    \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
      #2 8 * #3#4#5#6
      #1; #2
  }
\cs_new:Npn \__fp_trig_large_auxxi:w #1;
  {
    \exp_after:wN \__fp_ep_mul_raw:wwwwN
    \int_value:w \__fp_int_eval:w 0 \__fp_ep_to_ep_loop:N #1 ; ; !
```

```
    0,{7853}{9816}{3397}{4483}{0961}{5661};
    \__fp_trig_small:ww
  }
```

(*End of definition for* `\__fp_trig_large_auxvii:w` *and others.*)

77.1.6 Computing the power series

`\__fp_sin_series_o:NNwwww`
`\__fp_sin_series_aux_o:NNnwww`

Here we receive a conversion function `\__fp_ep_to_float_o:wwN` or `\__fp_ep_inv_to_float_o:wwN`, a ⟨*sign*⟩ (0 or 2), a (non-negative) ⟨*octant*⟩ delimited by a dot, a ⟨*fixed point*⟩ number delimited by a semicolon, and an extended-precision number. The auxiliary receives:

- the conversion function `#1`;
- the final sign, which depends on the octant `#3` and the sign `#2`;
- the octant `#3`, which controls the series we use;
- the square `#4 * #4` of the argument as a fixed point number, computed with `\__fp_fixed_mul:wwn`;
- the number itself as an extended-precision number.

If the octant is in $\{1,2,5,6,\dots\}$, we are near an extremum of the function and we use the series

$$\cos(x) = 1 - x^2\left(\frac{1}{2!} - x^2\left(\frac{1}{4!} - x^2\left(\cdots\right)\right)\right).$$

Otherwise, the series

$$\sin(x) = x\left(1 - x^2\left(\frac{1}{3!} - x^2\left(\frac{1}{5!} - x^2\left(\cdots\right)\right)\right)\right)$$

is used. Finally, the extended-precision number is converted to a floating point number with the given sign, and `\__fp_sanitize:Nw` checks for overflow and underflow.

```
\cs_new:Npn \__fp_sin_series_o:NNwwww #1#2#3. #4;
  {
    \__fp_fixed_mul:wwn #4; #4;
    {
      \exp_after:wN \__fp_sin_series_aux_o:NNnwww
      \exp_after:wN #1
      \int_value:w
        \if_int_odd:w \__fp_int_eval:w (#3 + 2) / 4 \__fp_int_eval_end:
          #2
        \else:
          \if_meaning:w #2 0 2 \else: 0 \fi:
        \fi:
      {#3}
    }
  }
\cs_new:Npn \__fp_sin_series_aux_o:NNnwww #1#2#3 #4; #5,#6;
  {
    \if_int_odd:w \__fp_int_eval:w #3 / 2 \__fp_int_eval_end:
      \exp_after:wN \use_i:nn
```

```
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
    { % 1/18!
      \__fp_fixed_mul_sub_back:wwwn    {0000}{0000}{0000}{0001}{5619}{2070};
                                    #4;{0000}{0000}{0000}{0477}{9477}{3324};
      \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0011}{4707}{4559}{7730};
      \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{2087}{6756}{9878}{6810};
      \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0027}{5573}{1922}{3985}{8907};
      \__fp_fixed_mul_sub_back:wwwn #4;{0000}{2480}{1587}{3015}{8730}{1587};
      \__fp_fixed_mul_sub_back:wwwn #4;{0013}{8888}{8888}{8888}{8888}{8889};
      \__fp_fixed_mul_sub_back:wwwn #4;{0416}{6666}{6666}{6666}{6666}{6667};
      \__fp_fixed_mul_sub_back:wwwn #4;{5000}{0000}{0000}{0000}{0000}{0000};
      \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
      { \__fp_fixed_continue:wn 0, }
    }
    { % 1/17!
      \__fp_fixed_mul_sub_back:wwwn    {0000}{0000}{0000}{0028}{1145}{7254};
                                    #4;{0000}{0000}{0000}{7647}{1637}{3182};
      \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0160}{5904}{3836}{8216};
      \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0002}{5052}{1083}{8544}{1719};
      \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0275}{5731}{9223}{9858}{9065};
      \__fp_fixed_mul_sub_back:wwwn #4;{0001}{9841}{2698}{4126}{9841}{2698};
      \__fp_fixed_mul_sub_back:wwwn #4;{0083}{3333}{3333}{3333}{3333}{3333};
      \__fp_fixed_mul_sub_back:wwwn #4;{1666}{6666}{6666}{6666}{6666}{6667};
      \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
      { \__fp_ep_mul:wwwwn 0, } #5,#6;
    }
    {
      \exp_after:wN \__fp_sanitize:Nw
      \exp_after:wN #2
      \int_value:w \__fp_int_eval:w #1
    }
    #2
  }
```

(*End of definition for `\__fp_sin_series_o:NNwwww` and `\__fp_sin_series_aux_o:NNnwww`.*)

`\__fp_tan_series_o:NNwwww`
`\__fp_tan_series_aux_o:Nnwww`

Contrarily to `\__fp_sin_series_o:NNwwww` which received a conversion auxiliary as `#1`, here, `#1` is 0 for tangent and 2 for cotangent. Consider first the case of the tangent. The octant `#3` starts at 1, which means that it is 1 or 2 for $|x| \in [0, \pi/2]$, it is 3 or 4 for $|x| \in [\pi/2, \pi]$, and so on: the intervals on which $\tan|x| \geq 0$ coincide with those for which $\lfloor(\#3 + 1)/2\rfloor$ is odd. We also have to take into account the original sign of x to get the sign of the final result; it is straightforward to check that the first `\int_value:w` expansion produces 0 for a positive final result, and 2 otherwise. A similar story holds for $\cot(x)$.

The auxiliary receives the sign, the octant, the square of the (reduced) input, and the (reduced) input (an extended-precision number) as arguments. It then computes the numerator and denominator of

$$\tan(x) \simeq \frac{x(1 - x^2(a_1 - x^2(a_2 - x^2(a_3 - x^2(a_4 - x^2 a_5)))))}{1 - x^2(b_1 - x^2(b_2 - x^2(b_3 - x^2(b_4 - x^2 b_5))))}.$$

The ratio is computed by `\__fp_ep_div:wwwwn`, then converted to a floating point number. For octants `#3` (really, quadrants) next to a pole of the functions, the fixed point numerator and denominator are exchanged before computing the ratio. Note that this `\if_int_odd:w` test relies on the fact that the octant is at least 1.

```
\cs_new:Npn \__fp_tan_series_o:NNwwww #1#2#3. #4;
  {
    \__fp_fixed_mul:wwn #4; #4;
    {
      \exp_after:wN \__fp_tan_series_aux_o:Nnwww
      \int_value:w
        \if_int_odd:w \__fp_int_eval:w #3 / 2 \__fp_int_eval_end:
          \exp_after:wN \reverse_if:N
        \fi:
        \if_meaning:w #1#2 2 \else: 0 \fi:
      {#3}
    }
  }
\cs_new:Npn \__fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5;
  {
    \__fp_fixed_mul_sub_back:wwwn     {0000}{0000}{1527}{3493}{0856}{7059};
                                 #3; {0000}{0159}{6080}{0274}{5257}{6472};
    \__fp_fixed_mul_sub_back:wwwn #3; {0002}{4571}{2320}{0157}{2558}{8481};
    \__fp_fixed_mul_sub_back:wwwn #3; {0115}{5830}{7533}{5397}{3168}{2147};
    \__fp_fixed_mul_sub_back:wwwn #3; {1929}{8245}{6140}{3508}{7719}{2982};
    \__fp_fixed_mul_sub_back:wwwn #3;{10000}{0000}{0000}{0000}{0000}{0000};
    { \__fp_ep_mul:wwwwn 0, } #4,#5;
    {
      \__fp_fixed_mul_sub_back:wwwn    {0000}{0007}{0258}{0681}{9408}{4706};
                                   #3;{0000}{2343}{7175}{1399}{6151}{7670};
      \__fp_fixed_mul_sub_back:wwwn #3;{0019}{2638}{4588}{9232}{8861}{3691};
      \__fp_fixed_mul_sub_back:wwwn #3;{0536}{6357}{0691}{4344}{6852}{4252};
      \__fp_fixed_mul_sub_back:wwwn #3;{5263}{1578}{9473}{6842}{1052}{6315};
      \__fp_fixed_mul_sub_back:wwwn#3;{10000}{0000}{0000}{0000}{0000}{0000};
      {
        \reverse_if:N \if_int_odd:w
            \__fp_int_eval:w (#2 - 1) / 2 \__fp_int_eval_end:
          \exp_after:wN \__fp_reverse_args:Nww
        \fi:
        \__fp_ep_div:wwwwn 0,
      }
    }
    {
      \exp_after:wN \__fp_sanitize:Nw
      \exp_after:wN #1
      \int_value:w \__fp_int_eval:w \__fp_ep_to_float_o:wwN
    }
    #1
  }
```

(*End of definition for* `\__fp_tan_series_o:NNwwww` *and* `\__fp_tan_series_aux_o:Nnwww`*.*)

77.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arccosecant, and arcsecant) are based on a function often denoted `atan2`. This function is accessed directly by feeding two arguments to arctangent, and is defined by $\operatorname{atan}(y,x) = \operatorname{atan}(y/x)$ for generic y and x. Its advantages over the conventional arctangent is that it takes values in $[-\pi,\pi]$ rather than $[-\pi/2,\pi/2]$, and that it is better behaved in boundary cases. Other inverse trigonometric functions are expressed in terms of atan as

$$\operatorname{acos} x = \operatorname{atan}(\sqrt{1-x^2}, x) \tag{5}$$
$$\operatorname{asin} x = \operatorname{atan}(x, \sqrt{1-x^2}) \tag{6}$$
$$\operatorname{asec} x = \operatorname{atan}(\sqrt{x^2-1}, 1) \tag{7}$$
$$\operatorname{acsc} x = \operatorname{atan}(1, \sqrt{x^2-1}) \tag{8}$$
$$\operatorname{atan} x = \operatorname{atan}(x, 1) \tag{9}$$
$$\operatorname{acot} x = \operatorname{atan}(1, x). \tag{10}$$

Rather than introducing a new function, `atan2`, the arctangent function `atan` is overloaded: it can take one or two arguments. In the comments below, following many texts, we call the first argument y and the second x, because $\operatorname{atan}(y,x) = \operatorname{atan}(y/x)$ is the angular coordinate of the point (x,y).

As for direct trigonometric functions, the first step in computing $\operatorname{atan}(y,x)$ is argument reduction. The sign of y gives that of the result. We distinguish eight regions where the point $(x,|y|)$ can lie, of angular size roughly $\pi/8$, characterized by their "octant", between 0 and 7 included. In each region, we compute an arctangent as a Taylor series, then shift this arctangent by the appropriate multiple of $\pi/4$ and sign to get the result. Here is a list of octants, and how we compute the arctangent (we assume $y > 0$: otherwise replace y by $-y$ below):

0 $0 < |y| < 0.41421x$, then $\operatorname{atan}\frac{|y|}{x}$ is given by a nicely convergent Taylor series;

1 $0 < 0.41421x < |y| < x$, then $\operatorname{atan}\frac{|y|}{x} = \frac{\pi}{4} - \operatorname{atan}\frac{x-|y|}{x+|y|}$;

2 $0 < 0.41421|y| < x < |y|$, then $\operatorname{atan}\frac{|y|}{x} = \frac{\pi}{4} + \operatorname{atan}\frac{-x+|y|}{x+|y|}$;

3 $0 < x < 0.41421|y|$, then $\operatorname{atan}\frac{|y|}{x} = \frac{\pi}{2} - \operatorname{atan}\frac{x}{|y|}$;

4 $0 < -x < 0.41421|y|$, then $\operatorname{atan}\frac{|y|}{x} = \frac{\pi}{2} + \operatorname{atan}\frac{-x}{|y|}$;

5 $0 < 0.41421|y| < -x < |y|$, then $\operatorname{atan}\frac{|y|}{x} = \frac{3\pi}{4} - \operatorname{atan}\frac{x+|y|}{-x+|y|}$;

6 $0 < -0.41421x < |y| < -x$, then $\operatorname{atan}\frac{|y|}{x} = \frac{3\pi}{4} + \operatorname{atan}\frac{-x-|y|}{-x+|y|}$;

7 $0 < |y| < -0.41421x$, then $\operatorname{atan}\frac{|y|}{x} = \pi - \operatorname{atan}\frac{|y|}{-x}$.

In the following, we denote by z the ratio among $|\frac{y}{x}|$, $|\frac{x}{y}|$, $|\frac{x+y}{x-y}|$, $|\frac{x-y}{x+y}|$ which appears in the right-hand side above.

77.2.1 Arctangent and arccotangent

`\__fp_atan_o:Nw`
`\__fp_acot_o:Nw`
`\__fp_atan_default:w`

The parsing step manipulates `atan` and `acot` like `min` and `max`, reading in an array of operands, but also leaves `\use_i:nn` or `\use_ii:nn` depending on whether the result should be given in radians or in degrees. The helper `\__fp_parse_function_one_two:nnw` checks that the operand is one or two floating point numbers (not tuples) and leaves its second argument or its tail accordingly (its first argument is used for error messages). More precisely if we are given a single floating point number `\__fp_atan_default:w` places `\c_one_fp` (expanded) after it; otherwise `\__fp_atan_default:w` is omitted by `\__fp_parse_function_one_two:nnw`.

```
\cs_new:Npn \__fp_atan_o:Nw #1
  {
    \__fp_parse_function_one_two:nnw
      { #1 { atan } { atand } }
      { \__fp_atan_default:w \__fp_atanii_o:Nww #1 }
  }
\cs_new:Npn \__fp_acot_o:Nw #1
  {
    \__fp_parse_function_one_two:nnw
      { #1 { acot } { acotd } }
      { \__fp_atan_default:w \__fp_acotii_o:Nww #1 }
  }
\cs_new:Npe \__fp_atan_default:w #1#2#3 @ { #1 #2 #3 \c_one_fp @ }
```

(*End of definition for* `\__fp_atan_o:Nw`*,* `\__fp_acot_o:Nw`*, and* `\__fp_atan_default:w`*.*)

`\__fp_atanii_o:Nww`
`\__fp_acotii_o:Nww`

If either operand is `nan`, we return it. If both are normal, we call `\__fp_atan_normal_o:NNnwNnw`. If both are zero or both infinity, we call `\__fp_atan_inf_o:NNNw` with argument 2, leading to a result among $\{\pm\pi/4, \pm 3\pi/4\}$ (in degrees, $\{\pm 45, \pm 135\}$). Otherwise, one is much bigger than the other, and we call `\__fp_atan_inf_o:NNNw` with either an argument of 4, leading to the values $\pm\pi/2$ (in degrees, ± 90), or 0, leading to $\{\pm 0, \pm\pi\}$ (in degrees, $\{\pm 0, \pm 180\}$). Since $\operatorname{acot}(x, y) = \operatorname{atan}(y, x)$, `\__fp_acotii_o:ww` simply reverses its two arguments.

```
\cs_new:Npn \__fp_atanii_o:Nww
    #1 \s__fp \__fp_chk:w #2#3#4; \s__fp \__fp_chk:w #5 #6 @
  {
    \if_meaning:w 3 #2 \__fp_case_return_i_o:ww \fi:
    \if_meaning:w 3 #5 \__fp_case_return_ii_o:ww \fi:
    \if_case:w
      \if_meaning:w #2 #5
        \if_meaning:w 1 #2 10 \else: 0 \fi:
      \else:
        \if_int_compare:w #2 > #5 \exp_stop_f: 1 \else: 2 \fi:
      \fi:
      \exp_stop_f:
         \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 2 }
    \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 4 }
    \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 0 }
    \fi:
    \__fp_atan_normal_o:NNnwNnw #1
    \s__fp \__fp_chk:w #2#3#4;
    \s__fp \__fp_chk:w #5 #6
  }
\cs_new:Npn \__fp_acotii_o:Nww #1#2; #3;
  { \__fp_atanii_o:Nww #1#3; #2; }
```

(*End of definition for* `\__fp_atanii_o:Nww` *and* `\__fp_acotii_o:Nww`*.*)

`\__fp_atan_inf_o:NNNw` This auxiliary is called whenever one number is ± 0 or $\pm\infty$ (and neither is `nan`). Then the result only depends on the signs, and its value is a multiple of $\pi/4$. We use the same auxiliary as for normal numbers, `\__fp_atan_combine_o:NwwwwwN`, with arguments the final sign `#2`; the octant `#3`; $\operatorname{atan} z/z = 1$ as a fixed point number; $z = 0$ as a fixed point number; and $z = 0$ as an extended-precision number. Given the values we provide, $\operatorname{atan} z$ is computed to be 0, and the result is $[\texttt{#3}/2] \cdot \pi/4$ if the sign `#5` of x is positive, and $[(7 - \texttt{#3})/2] \cdot \pi/4$ for negative x, where the divisions are rounded up.

```
\cs_new:Npn \__fp_atan_inf_o:NNNw #1#2#3 \s__fp \__fp_chk:w #4#5#6;
  {
    \exp_after:wN \__fp_atan_combine_o:NwwwwwN
    \exp_after:wN #2
    \int_value:w \__fp_int_eval:w
      \if_meaning:w 2 #5 7 - \fi: #3 \exp_after:wN ;
    \c__fp_one_fixed_tl
    {0000}{0000}{0000}{0000}{0000}{0000};
    0,{0000}{0000}{0000}{0000}{0000}{0000}; #1
  }
```

(*End of definition for* `\__fp_atan_inf_o:NNNw`.)

`\__fp_atan_normal_o:NNnwNnw` Here we simply reorder the floating point data into a pair of signed extended-precision numbers, that is, a sign, an exponent ending with a comma, and a six-block mantissa ending with a semi-colon. This extended precision is required by other inverse trigonometric functions, to compute things like $\operatorname{atan}(x, \sqrt{1-x^2})$ without intermediate rounding errors.

```
\cs_new_protected:Npn \__fp_atan_normal_o:NNnwNnw
    #1 \s__fp \__fp_chk:w 1#2#3#4; \s__fp \__fp_chk:w 1#5#6#7;
  {
    \__fp_atan_test_o:NwwNwwN
      #2 #3, #4{0000}{0000};
      #5 #6, #7{0000}{0000}; #1
  }
```

(*End of definition for* `\__fp_atan_normal_o:NNnwNnw`.)

`\__fp_atan_test_o:NwwNwwN` This receives: the sign `#1` of y, its exponent `#2`, its 24 digits `#3` in groups of 4, and similarly for x. We prepare to call `\__fp_atan_combine_o:NwwwwwN` which expects the sign `#1`, the octant, the ratio $(\operatorname{atan} z)/z = 1 - \cdots$, and the value of z, both as a fixed point number and as an extended-precision floating point number with a mantissa in $[0.01, 1)$. For now, we place `#1` as a first argument, and start an integer expression for the octant. The sign of x does not affect z, so we simply leave a contribution to the octant: $\langle\mathit{octant}\rangle \to 7 - \langle\mathit{octant}\rangle$ for negative x. Then we order $|y|$ and $|x|$ in a non-decreasing order: if $|y| > |x|$, insert $3-$ in the expression for the octant, and swap the two numbers. The finer test with 0.41421 is done by `\__fp_atan_div:wnwwnw` after the operands have been ordered.

```
\cs_new:Npn \__fp_atan_test_o:NwwNwwN #1#2,#3; #4#5,#6;
  {
    \exp_after:wN \__fp_atan_combine_o:NwwwwwN
    \exp_after:wN #1
    \int_value:w \__fp_int_eval:w
      \if_meaning:w 2 #4
        7 - \__fp_int_eval:w
      \fi:
```

```
      \if_int_compare:w
          \__fp_ep_compare:wwww #2,#3; #5,#6; > \c_zero_int
        3 -
        \exp_after:wN \__fp_reverse_args:Nww
      \fi:
      \__fp_atan_div:wnwwnw #2,#3; #5,#6;
  }
```

(*End of definition for* `\__fp_atan_test_o:NwwNwwN`.)

`\__fp_atan_div:wnwwnw` `\__fp_atan_near:wwwn` `\__fp_atan_near_aux:wwn`

This receives two positive numbers a and b (equal to $|x|$ and $|y|$ in some order), each as an exponent and 6 blocks of 4 digits, such that $0 < a < b$. If $0.41421b < a$, the two numbers are "near", hence the point (y, x) that we started with is closer to the diagonals $\{|y| = |x|\}$ than to the axes $\{xy = 0\}$. In that case, the octant is 1 (possibly combined with the $7-$ and $3-$ inserted earlier) and we wish to compute atan $\frac{b-a}{a+b}$. Otherwise, the octant is 0 (again, combined with earlier terms) and we wish to compute atan $\frac{a}{b}$. In any case, call `\__fp_atan_auxi:ww` followed by z, as a comma-delimited exponent and a fixed point number.

```
\cs_new:Npn \__fp_atan_div:wnwwnw #1,#2#3; #4,#5#6;
  {
    \if_int_compare:w
      \__fp_int_eval:w 41421 * #5 < #2 000
        \if_case:w \__fp_int_eval:w #4 - #1 \__fp_int_eval_end:
          00 \or: 0 \fi:
      \exp_stop_f:
      \exp_after:wN \__fp_atan_near:wwwn
    \fi:
    0
    \__fp_ep_div:wwwwn #1,{#2}#3; #4,{#5}#6;
    \__fp_atan_auxi:ww
  }
\cs_new:Npn \__fp_atan_near:wwwn
    0 \__fp_ep_div:wwwwn #1,#2; #3,
  {
    1
    \__fp_ep_to_fixed:wwn #1 - #3, #2;
    \__fp_atan_near_aux:wwn
  }
\cs_new:Npn \__fp_atan_near_aux:wwn #1; #2;
  {
    \__fp_fixed_add:wwn #1; #2;
    { \__fp_fixed_sub:wwn #2; #1; { \__fp_ep_div:wwwwn 0, } 0, }
  }
```

(*End of definition for* `\__fp_atan_div:wnwwnw`, `\__fp_atan_near:wwwn`, *and* `\__fp_atan_near_aux:wwn`.)

`\__fp_atan_auxi:ww` `\__fp_atan_auxii:w`

Convert z from a representation as an exponent and a fixed point number in $[0.01, 1)$ to a fixed point number only, then set up the call to `\__fp_atan_Taylor_loop:www`, followed by the fixed point representation of z and the old representation.

```
\cs_new:Npn \__fp_atan_auxi:ww #1,#2;
  { \__fp_ep_to_fixed:wwn #1,#2; \__fp_atan_auxii:w #1,#2; }
\cs_new:Npn \__fp_atan_auxii:w #1;
  {
    \__fp_fixed_mul:wwn #1; #1;
```

```
    {
      \__fp_atan_Taylor_loop:www 39 ;
        {0000}{0000}{0000}{0000}{0000}{0000} ;
    }
    ! #1;
  }
```

(*End of definition for* `\__fp_atan_auxi:ww` *and* `\__fp_atan_auxii:w`.)

`\__fp_atan_Taylor_loop:www`
`\__fp_atan_Taylor_break:w`

We compute the series of $(\operatorname{atan} z)/z$. A typical intermediate stage has `#1` $= 2k-1$, `#2` $= \frac{1}{2k+1} - z^2(\frac{1}{2k+3} - z^2(\cdots - z^2\frac{1}{39}))$, and `#3` $= z^2$. To go to the next step $k \to k-1$, we compute $\frac{1}{2k-1}$, then subtract from it z^2 times `#2`. The loop stops when $k = 0$: then `#2` is $(\operatorname{atan} z)/z$, and there is a need to clean up all the unnecessary data, end the integer expression computing the octant with a semicolon, and leave the result `#2` afterwards.

```
\cs_new:Npn \__fp_atan_Taylor_loop:www #1; #2; #3;
  {
    \if_int_compare:w #1 = - \c_one_int
      \__fp_atan_Taylor_break:w
    \fi:
    \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl #1;
    \__fp_rrot:www \__fp_fixed_mul_sub_back:wwwn #2; #3;
    {
      \exp_after:wN \__fp_atan_Taylor_loop:www
      \int_value:w \__fp_int_eval:w #1 - 2 ;
    }
    #3;
  }
\cs_new:Npn \__fp_atan_Taylor_break:w
    \fi: #1 \__fp_fixed_mul_sub_back:wwwn #2; #3 !
  { \fi: ; #2 ; }
```

(*End of definition for* `\__fp_atan_Taylor_loop:www` *and* `\__fp_atan_Taylor_break:w`.)

`\__fp_atan_combine_o:NwwwwwN`
`\__fp_atan_combine_aux:ww`

This receives a ⟨*sign*⟩, an ⟨*octant*⟩, a fixed point value of $(\operatorname{atan} z)/z$, a fixed point number z, and another representation of z, as an ⟨*exponent*⟩ and the fixed point number $10^{-\langle\mathit{exponent}\rangle} z$, followed by either `\use_i:nn` (when working in radians) or `\use_ii:nn` (when working in degrees). The function computes the floating point result

$$\langle\mathit{sign}\rangle \left(\left\lceil \frac{\langle\mathit{octant}\rangle}{2} \right\rceil \frac{\pi}{4} + (-1)^{\langle\mathit{octant}\rangle} \frac{\operatorname{atan} z}{z} \cdot z \right) , \tag{11}$$

multiplied by $180/\pi$ if working in degrees, and using in any case the most appropriate representation of z. The floating point result is passed to `\__fp_sanitize:Nw`, which checks for overflow or underflow. If the octant is 0, leave the exponent `#5` for `\__fp_sanitize:Nw`, and multiply `#3` $= \frac{\operatorname{atan} z}{z}$ with `#6`, the adjusted z. Otherwise, multiply `#3` $= \frac{\operatorname{atan} z}{z}$ with `#4` $= z$, then compute the appropriate multiple of $\frac{\pi}{4}$ and add or subtract the product `#3` $\cdot$ `#4`. In both cases, convert to a floating point with `\__fp_fixed_to_float_o:wN`.

```
\cs_new:Npn \__fp_atan_combine_o:NwwwwwN #1 #2; #3; #4; #5,#6; #7
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN #1
    \int_value:w \__fp_int_eval:w
```

```
      \if_meaning:w 0 #2
        \exp_after:wN \use_i:nn
      \else:
        \exp_after:wN \use_ii:nn
      \fi:
      { #5 \__fp_fixed_mul:wwn #3; #6; }
      {
        \__fp_fixed_mul:wwn #3; #4;
        {
          \exp_after:wN \__fp_atan_combine_aux:ww
          \int_value:w \__fp_int_eval:w #2 / 2 ; #2;
        }
      }
      { #7 \__fp_fixed_to_float_o:wN \__fp_fixed_to_float_rad_o:wN }
      #1
  }
\cs_new:Npn \__fp_atan_combine_aux:ww #1; #2;
  {
    \__fp_fixed_mul_short:wwn
      {7853}{9816}{3397}{4483}{0961}{5661};
      {#1}{0000}{0000};
    {
      \if_int_odd:w #2 \exp_stop_f:
        \exp_after:wN \__fp_fixed_sub:wwn
      \else:
        \exp_after:wN \__fp_fixed_add:wwn
      \fi:
    }
  }
```

(*End of definition for* `\__fp_atan_combine_o:NwwwwwN` *and* `\__fp_atan_combine_aux:ww`.)

77.2.2 Arcsine and arccosine

`\__fp_asin_o:w` Again, the first argument provided by l3fp-parse is `\use_i:nn` if we are to work in radians and `\use_ii:nn` for degrees. Then comes a floating point number. The arcsine of ± 0 or `nan` is the same floating point number. The arcsine of $\pm\infty$ raises an invalid operation exception. Otherwise, call an auxiliary common with `\__fp_acos_o:w`, feeding it information about what function is being performed (for "invalid operation" exceptions).

```
\cs_new:Npn \__fp_asin_o:w #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_case:w #2 \exp_stop_f:
      \__fp_case_return_same_o:w
    \or:
      \__fp_case_use:nw
        { \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { asin } { asind } } }
    \or:
      \__fp_case_use:nw
        { \__fp_invalid_operation_o:fw { #1 { asin } { asind } } }
    \else:
      \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3;
  }
```

(*End of definition for* `\__fp_asin_o:w`.)

`\__fp_acos_o:w` The arccosine of ± 0 is $\pi/2$ (in degrees, 90). The arccosine of $\pm\infty$ raises an invalid operation exception. The arccosine of `nan` is itself. Otherwise, call an auxiliary common with `\__fp_sin_o:w`, informing it that it was called by `acos` or `acosd`, and preparing to swap some arguments down the line.

```
\cs_new:Npn \__fp_acos_o:w #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_case:w #2 \exp_stop_f:
      \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 4 }
    \or:
      \__fp_case_use:nw
        {
          \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { acos } { acosd } }
            \__fp_reverse_args:Nww
        }
    \or:
      \__fp_case_use:nw
        { \__fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
    \else:
      \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3;
  }
```

(*End of definition for* `\__fp_acos_o:w`.)

`\__fp_asin_normal_o:NfwNnnnnw` If the exponent `#5` is at most 0, the operand lies within $(-1,1)$ and the operation is permitted: call `\__fp_asin_auxi_o:NnNww` with the appropriate arguments. If the number is exactly ± 1 (the test works because we know that `#5` ≥ 1, `#6#7` ≥ 10000000, `#8#9` ≥ 0, with equality only for ± 1), we also call `\__fp_asin_auxi_o:NnNww`. Otherwise, `\__fp_use_i:ww` gets rid of the `asin` auxiliary, and raises instead an invalid operation, because the operand is outside the domain of arcsine or arccosine.

```
\cs_new:Npn \__fp_asin_normal_o:NfwNnnnnw
    #1#2#3 \s__fp \__fp_chk:w 1#4#5#6#7#8#9;
  {
    \if_int_compare:w #5 < \c_one_int
      \exp_after:wN \__fp_use_none_until_s:w
    \fi:
    \if_int_compare:w \__fp_int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
      \exp_after:wN \__fp_use_none_until_s:w
    \fi:
    \__fp_use_i:ww
    \__fp_invalid_operation_o:fw {#2}
      \s__fp \__fp_chk:w 1#4{#5}{#6}{#7}{#8}{#9};
    \__fp_asin_auxi_o:NnNww
      #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000};
  }
```

(*End of definition for* `\__fp_asin_normal_o:NfwNnnnnw`.)

`\__fp_asin_auxi_o:NnNww` `\__fp_asin_isqrt:wn` We compute $x/\sqrt{1-x^2}$. This function is used by `asin` and `acos`, but also by `acsc` and `asec` after inverting the operand, thus it must manipulate extended-precision numbers. First evaluate $1-x^2$ as $(1+x)(1-x)$: this behaves better near $x=1$. We do the addition/subtraction with fixed point numbers (they are not implemented for extended-precision floats), but go back to extended-precision floats to multiply and compute the inverse square root $1/\sqrt{1-x^2}$. Finally, multiply by the (positive) extended-precision float $|x|$, and feed the (signed) result, and the number $+1$, as arguments to the arctangent function. When computing the arccosine, the arguments $x/\sqrt{1-x^2}$ and $+1$ are swapped by `#2` (`\__fp_reverse_args:Nww` in that case) before `\__fp_atan_test_o:NwwNwwN` is evaluated. Note that the arctangent function requires normalized arguments, hence the need for `ep_to_ep` and `continue` after `ep_mul`.

```
\cs_new:Npn \__fp_asin_auxi_o:NnNww #1#2#3#4,#5;
  {
    \__fp_ep_to_fixed:wwn #4,#5;
    \__fp_asin_isqrt:wn
    \__fp_ep_mul:wwwwn #4,#5;
    \__fp_ep_to_ep:wwN
    \__fp_fixed_continue:wn
    { #2 \__fp_atan_test_o:NwwNwwN #3 }
    0 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1
  }
\cs_new:Npn \__fp_asin_isqrt:wn #1;
  {
    \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl #1;
    {
      \__fp_fixed_add_one:wN #1;
      \__fp_fixed_continue:wn { \__fp_ep_mul:wwwwn 0, } 0,
    }
    \__fp_ep_isqrt:wwn
  }
```

(*End of definition for* `\__fp_asin_auxi_o:NnNww` *and* `\__fp_asin_isqrt:wn`.)

77.2.3 Arccosecant and arcsecant

`\__fp_acsc_o:w` Cases are mostly labelled by `#2`, except when `#2` is 2: then we use `#3#2`, which is $02 = 2$ when the number is $+\infty$ and 22 when the number is $-\infty$. The arccosecant of ± 0 raises an invalid operation exception. The arccosecant of $\pm\infty$ is ± 0 with the same sign. The arcosecant of `nan` is itself. Otherwise, `\__fp_acsc_normal_o:NfwNnw` does some more tests, keeping the function name (`acsc` or `acscd`) as an argument for invalid operation exceptions.

```
\cs_new:Npn \__fp_acsc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
           \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { acsc } { acscd } } }
    \or:   \__fp_case_use:nw
             { \__fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acscd } } }
    \or:   \__fp_case_return_o:Nw \c_zero_fp
    \or:   \__fp_case_return_same_o:w
    \else: \__fp_case_return_o:Nw \c_minus_zero_fp
    \fi:
    \s__fp \__fp_chk:w #2 #3 #4;
  }
```

(*End of definition for* `\__fp_acsc_o:w`.)

`\__fp_asec_o:w` The arcsecant of ± 0 raises an invalid operation exception. The arcsecant of $\pm\infty$ is $\pi/2$ (in degrees, 90). The arcosecant of `nan` is itself. Otherwise, do some more tests, keeping the function name `asec` (or `asecd`) as an argument for invalid operation exceptions, and a `\__fp_reverse_args:Nww` following precisely that appearing in `\__fp_acos_o:w`.

```
\cs_new:Npn \__fp_asec_o:w #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_case:w #2 \exp_stop_f:
      \__fp_case_use:nw
        { \__fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
    \or:
      \__fp_case_use:nw
        {
          \__fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
            \__fp_reverse_args:Nww
        }
    \or:   \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 4 }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3;
  }
```

(*End of definition for* `\__fp_asec_o:w`.)

`\__fp_acsc_normal_o:NfwNnw` If the exponent is non-positive, the operand is less than 1 in absolute value, which is always an invalid operation: complain. Otherwise, compute the inverse of the operand, and feed it to `\__fp_asin_auxi_o:NnNww` (with all the appropriate arguments). This computes what we want thanks to $\operatorname{acsc}(x) = \operatorname{asin}(1/x)$ and $\operatorname{asec}(x) = \operatorname{acos}(1/x)$.

```
\cs_new:Npn \__fp_acsc_normal_o:NfwNnw #1#2#3 \s__fp \__fp_chk:w 1#4#5#6;
  {
    \int_compare:nNnTF {#5} < 1
      {
        \__fp_invalid_operation_o:fw {#2}
          \s__fp \__fp_chk:w 1#4{#5}#6;
      }
      {
        \__fp_ep_div:wwwwn
          1,{1000}{0000}{0000}{0000}{0000}{0000};
          #5,#6{0000}{0000};
        { \__fp_asin_auxi_o:NnNww #1 {#3} #4 }
      }
  }
```

(*End of definition for* `\__fp_acsc_normal_o:NfwNnw`.)

```
⟨/package⟩
```

Chapter 78

l3fp-convert implementation

```
⟨*package⟩
⟨@@=fp⟩
```

78.1 Dealing with tuples

__fp_tuple_convert:Nw
__fp_tuple_convert_loop:nNw
__fp_tuple_convert_end:w

The first argument is for instance `\__fp_to_tl_dispatch:w`, which converts any floating point object to the appropriate representation. We loop through all items, putting `,~` between all of them and making sure to remove the leading `,~`.

```
\cs_new:Npn \__fp_tuple_convert:Nw #1 \s__fp_tuple \__fp_tuple_chk:w #2 ;
  {
    \int_case:nnF { \__fp_array_count:n {#2} }
      {
        { 0 } { ( ) }
        { 1 } { \__fp_tuple_convert_end:w @ { #1 #2 , } }
      }
      {
        \__fp_tuple_convert_loop:nNw { } #1
          #2 { ? \__fp_tuple_convert_end:w } ;
          @ { \use_none:nn }
      }
  }
\cs_new:Npn \__fp_tuple_convert_loop:nNw #1#2#3#4; #5 @ #6
  {
    \use_none:n #3
    \exp_args:Nf \__fp_tuple_convert_loop:nNw { #2 #3#4 ; } #2 #5
      @ { #6 , ~ #1 }
  }
\cs_new:Npn \__fp_tuple_convert_end:w #1 @ #2
  { \exp_after:wN ( \exp:w \exp_end_continue_f:w #2 ) }
```

(*End of definition for `\__fp_tuple_convert:Nw`, `\__fp_tuple_convert_loop:nNw`, and `\__fp_tuple_convert_end:w`.*)

78.2 Trimming trailing zeros

__fp_trim_zeros:w
__fp_trim_zeros_loop:w
__fp_trim_zeros_dot:w
__fp_trim_zeros_end:w

If `#1` ends with a 0, the `loop` auxiliary takes that zero as an end-delimiter for its first argument, and the second argument is the same `loop` auxiliary. Once the last trailing zero is reached, the second argument is the dot auxiliary, which removes a trailing dot if any. We then clean-up with the end auxiliary, keeping only the number.

```
\cs_new:Npn \__fp_trim_zeros:w #1 ;
  {
    \__fp_trim_zeros_loop:w #1
      ; \__fp_trim_zeros_loop:w 0; \__fp_trim_zeros_dot:w .; \s__fp_stop
  }
\cs_new:Npn \__fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
\cs_new:Npn \__fp_trim_zeros_dot:w #1 .; { \__fp_trim_zeros_end:w #1 ; }
\cs_new:Npn \__fp_trim_zeros_end:w #1 ; #2 \s__fp_stop { #1 }
```

(*End of definition for* `\__fp_trim_zeros:w` *and others.*)

78.3 Scientific notation

`\fp_to_scientific:N`
`\fp_to_scientific:c`
`\fp_to_scientific:n`

The three public functions evaluate their argument, then pass it to `\__fp_to_scientific_dispatch:w`.

```
\cs_new:Npn \fp_to_scientific:N #1
  { \exp_after:wN \__fp_to_scientific_dispatch:w #1 }
\cs_generate_variant:Nn \fp_to_scientific:N { c }
\cs_new:Npn \fp_to_scientific:n
  {
    \exp_after:wN \__fp_to_scientific_dispatch:w
    \exp:w \exp_end_continue_f:w \__fp_parse:n
  }
```

(*End of definition for* `\fp_to_scientific:N` *and* `\fp_to_scientific:n`*. These functions are documented on page* 261*.*)

`\__fp_to_scientific_dispatch:w`
`\__fp_to_scientific_recover:w`
`\__fp_tuple_to_scientific:w`

We allow tuples.

```
\cs_new:Npn \__fp_to_scientific_dispatch:w #1
  {
    \__fp_change_func_type:NNN
      #1 \__fp_to_scientific:w \__fp_to_scientific_recover:w
    #1
  }
\cs_new:Npn \__fp_to_scientific_recover:w #1 #2 ;
  {
    \__fp_error:nffn { unknown-type } { \tl_to_str:n { #2 ; } } { } { }
    nan
  }
\cs_new:Npn \__fp_tuple_to_scientific:w
  { \__fp_tuple_convert:Nw \__fp_to_scientific_dispatch:w }
```

(*End of definition for* `\__fp_to_scientific_dispatch:w`*,* `\__fp_to_scientific_recover:w`*, and* `\__fp_tuple_to_scientific:w`*.*)

`\__fp_to_scientific:w`
`\__fp_to_scientific_normal:wnnnnn`
`\__fp_to_scientific_normal:wNw`

Expressing an internal floating point number in scientific notation is quite easy: no rounding, and the format is very well defined. First cater for the sign: negative numbers ($\#2 = 2$) start with `-`; we then only need to care about positive numbers and `nan`. Then filter the special cases: ± 0 are represented as `0`; infinities are converted to a number slightly larger than the largest after an "invalid_operation" exception; `nan` is represented as `0` after an "invalid_operation" exception. In the normal case, decrement the exponent and unbrace the 4 brace groups, then in a second step grab the first digit (previously hidden in braces) to order the various parts correctly.

```
\cs_new:Npn \__fp_to_scientific:w \s__fp \__fp_chk:w #1#2
  {
    \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
    \if_case:w #1 \exp_stop_f:
         \__fp_case_return:nw { 0.000000000000000e0 }
    \or: \exp_after:wN \__fp_to_scientific_normal:wnnnnn
    \or:
      \__fp_case_use:nw
        {
          \__fp_invalid_operation:nnw
            { \fp_to_scientific:N \c__fp_overflowing_fp }
            { fp_to_scientific }
        }
    \or:
      \__fp_case_use:nw
        {
          \__fp_invalid_operation:nnw
            { \fp_to_scientific:N \c_zero_fp }
            { fp_to_scientific }
        }
    \fi:
    \s__fp \__fp_chk:w #1 #2
  }
\cs_new:Npn \__fp_to_scientific_normal:wnnnnn
  \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
  {
    \exp_after:wN \__fp_to_scientific_normal:wNw
    \exp_after:wN e
    \int_value:w \__fp_int_eval:w #2 - 1
    ; #3 #4 #5 #6 ;
  }
\cs_new:Npn \__fp_to_scientific_normal:wNw #1 ; #2#3;
  { #2.#3 #1 }
```

(*End of definition for* `\__fp_to_scientific:w`, `\__fp_to_scientific_normal:wnnnnn`, *and* `\__fp_to_scientific_normal:wNw`.)

78.4 Decimal representation

`\fp_to_decimal:N`
`\fp_to_decimal:c`
`\fp_to_decimal:n`

All three public variants are based on the same `\__fp_to_decimal_dispatch:w` after evaluating their argument to an internal floating point.

```
\cs_new:Npn \fp_to_decimal:N #1
  { \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
\cs_generate_variant:Nn \fp_to_decimal:N { c }
\cs_new:Npn \fp_to_decimal:n
  {
    \exp_after:wN \__fp_to_decimal_dispatch:w
    \exp:w \exp_end_continue_f:w \__fp_parse:n
  }
```

(*End of definition for* `\fp_to_decimal:N` *and* `\fp_to_decimal:n`. *These functions are documented on page* *260*.)

__fp_to_decimal_dispatch:w
__fp_to_decimal_recover:w
__fp_tuple_to_decimal:w

We allow tuples.

```
\cs_new:Npn \__fp_to_decimal_dispatch:w #1
  {
    \__fp_change_func_type:NNN
      #1 \__fp_to_decimal:w \__fp_to_decimal_recover:w
    #1
  }
\cs_new:Npn \__fp_to_decimal_recover:w #1 #2 ;
  {
    \__fp_error:nffn { unknown-type } { \tl_to_str:n { #2 ; } } { } { }
    nan
  }
\cs_new:Npn \__fp_tuple_to_decimal:w
  { \__fp_tuple_convert:Nw \__fp_to_decimal_dispatch:w }
```

(*End of definition for* `\__fp_to_decimal_dispatch:w`*,* `\__fp_to_decimal_recover:w`*, and* `\__fp_tuple_to_decimal:w`*.*)

__fp_to_decimal:w
__fp_to_decimal_normal:wnnnnn
__fp_to_decimal_large:Nnnw
__fp_to_decimal_huge:wnnnn

The structure is similar to `\__fp_to_scientific:w`. Insert `-` for negative numbers. Zero gives 0, $\pm\infty$ and `nan` yield an "invalid operation" exception; note that $\pm\infty$ produces a very large output, which we don't expand now since it most likely won't be needed. Normal numbers with an exponent in the range $[1,15]$ have that number of digits before the decimal separator: "decimate" them, and remove leading zeros with `\int_value:w`, then trim trailing zeros and dot. Normal numbers with an exponent 16 or larger have no decimal separator, we only need to add trailing zeros. When the exponent is non-positive, the result should be 0.⟨*zeros*⟩⟨*digits*⟩, trimmed.

```
\cs_new:Npn \__fp_to_decimal:w \s__fp \__fp_chk:w #1#2
  {
    \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
    \if_case:w #1 \exp_stop_f:
         \__fp_case_return:nw { 0 }
    \or: \exp_after:wN \__fp_to_decimal_normal:wnnnnn
    \or:
      \__fp_case_use:nw
        {
          \__fp_invalid_operation:nnw
            { \fp_to_decimal:N \c__fp_overflowing_fp }
            { fp_to_decimal }
        }
    \or:
      \__fp_case_use:nw
        {
          \__fp_invalid_operation:nnw
            { 0 }
            { fp_to_decimal }
        }
    \fi:
    \s__fp \__fp_chk:w #1 #2
  }
\cs_new:Npn \__fp_to_decimal_normal:wnnnnn
    \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
  {
    \int_compare:nNnTF {#2} > 0
      {
```

```
        \int_compare:nNnTF {#2} < \c__fp_prec_int
          {
            \__fp_decimate:nNnnnn { \c__fp_prec_int - #2 }
              \__fp_to_decimal_large:Nnnw
          }
          {
            \exp_after:wN \exp_after:wN
            \exp_after:wN \__fp_to_decimal_huge:wnnnn
            \prg_replicate:nn { #2 - \c__fp_prec_int } { 0 } ;
          }
        {#3} {#4} {#5} {#6}
      }
      {
        \exp_after:wN \__fp_trim_zeros:w
        \exp_after:wN 0
        \exp_after:wN .
        \exp:w \exp_end_continue_f:w \prg_replicate:nn { - #2 } { 0 }
        #3#4#5#6 ;
      }
  }
\cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4;
  {
    \exp_after:wN \__fp_trim_zeros:w \int_value:w
      \if_int_compare:w #2 > \c_zero_int
        #2
      \fi:
      \exp_stop_f:
      #3.#4 ;
  }
\cs_new:Npn \__fp_to_decimal_huge:wnnnn #1; #2#3#4#5 { #2#3#4#5 #1 }
```

(*End of definition for* `\__fp_to_decimal:w` *and others.*)

78.5 Token list representation

`\fp_to_tl:N` `\fp_to_tl:c` `\fp_to_tl:n`

These three public functions evaluate their argument, then pass it to `\__fp_to_tl_dispatch:w`.

```
\cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \__fp_to_tl_dispatch:w #1 }
\cs_generate_variant:Nn \fp_to_tl:N { c }
\cs_new:Npn \fp_to_tl:n
  {
    \exp_after:wN \__fp_to_tl_dispatch:w
    \exp:w \exp_end_continue_f:w \__fp_parse:n
  }
```

(*End of definition for* `\fp_to_tl:N` *and* `\fp_to_tl:n`*. These functions are documented on page 261.*)

`\__fp_to_tl_dispatch:w` `\__fp_to_tl_recover:w` `\__fp_tuple_to_tl:w`

We allow tuples.

```
\cs_new:Npn \__fp_to_tl_dispatch:w #1
  { \__fp_change_func_type:NNN #1 \__fp_to_tl:w \__fp_to_tl_recover:w #1 }
\cs_new:Npn \__fp_to_tl_recover:w #1 #2 ;
  {
    \__fp_error:nffn { unknown-type } { \tl_to_str:n { #2 ; } } { } { }
```

```
    nan
  }
\cs_new:Npn \__fp_tuple_to_tl:w
  { \__fp_tuple_convert:Nw \__fp_to_tl_dispatch:w }
```

(*End of definition for* `\__fp_to_tl_dispatch:w`*,* `\__fp_to_tl_recover:w`*, and* `\__fp_tuple_to_tl:w`*.*)

`\__fp_to_tl:w`
`\__fp_to_tl_normal:nnnnn`
`\__fp_to_tl_scientific:wnnnnn`
`\__fp_to_tl_scientific:wNw`

A structure similar to `\__fp_to_scientific_dispatch:w` and `\__fp_to_decimal_dispatch:w`, but without the "invalid operation" exception. First filter special cases. We express normal numbers in decimal notation if the exponent is in the range $[-2, 16]$, and otherwise use scientific notation.

```
\cs_new:Npn \__fp_to_tl:w \s__fp \__fp_chk:w #1#2
  {
    \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
    \if_case:w #1 \exp_stop_f:
           \__fp_case_return:nw { 0 }
    \or:   \exp_after:wN \__fp_to_tl_normal:nnnnn
    \or:   \__fp_case_return:nw { inf }
    \else: \__fp_case_return:nw { nan }
    \fi:
  }
\cs_new:Npn \__fp_to_tl_normal:nnnnn #1
  {
    \int_compare:nTF
      { -2 <= #1 <= \c__fp_prec_int }
      { \__fp_to_decimal_normal:wnnnnn }
      { \__fp_to_tl_scientific:wnnnnn }
    \s__fp \__fp_chk:w 1 0 {#1}
  }
\cs_new:Npn \__fp_to_tl_scientific:wnnnnn
  \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
  {
    \exp_after:wN \__fp_to_tl_scientific:wNw
    \exp_after:wN e
    \int_value:w \__fp_int_eval:w #2 - 1
    ; #3 #4 #5 #6 ;
  }
\cs_new:Npn \__fp_to_tl_scientific:wNw #1 ; #2#3;
  { \__fp_trim_zeros:w #2.#3 ; #1 }
```

(*End of definition for* `\__fp_to_tl:w` *and others.*)

78.6 Formatting

This is not implemented yet, as it is not yet clear what a correct interface would be, for this kind of structured conversion from a floating point (or other types of variables) to a string. Ideas welcome.

78.7 Convert to dimension or integer

`\fp_to_dim:N`
`\fp_to_dim:c`
`\fp_to_dim:n`
`\__fp_to_dim_dispatch:w`
`\__fp_to_dim_recover:w`
`\__fp_to_dim:w`

All three public variants are based on the same `\__fp_to_dim_dispatch:w` after evaluating their argument to an internal floating point. We only allow floating point numbers, not tuples.

```
\cs_new:Npn \fp_to_dim:N #1
  { \exp_after:wN \__fp_to_dim_dispatch:w #1 }
\cs_generate_variant:Nn \fp_to_dim:N { c }
\cs_new:Npn \fp_to_dim:n
  {
    \exp_after:wN \__fp_to_dim_dispatch:w
    \exp:w \exp_end_continue_f:w \__fp_parse:n
  }
\cs_new:Npn \__fp_to_dim_dispatch:w #1#2 ;
  {
    \__fp_change_func_type:NNN #1 \__fp_to_dim:w \__fp_to_dim_recover:w
    #1 #2 ;
  }
\cs_new:Npn \__fp_to_dim_recover:w #1
  { \__fp_invalid_operation:nnw { 0pt } { fp_to_dim } }
\cs_new:Npn \__fp_to_dim:w #1 ; { \__fp_to_decimal:w #1 ; pt }
```

(*End of definition for* `\fp_to_dim:N` *and others. These functions are documented on page 260.*)

`\fp_to_int:N`
`\fp_to_int:c`
`\fp_to_int:n`
`\__fp_to_int_dispatch:w`
`\__fp_to_int_recover:w`

For the most part identical to `\fp_to_dim:N` but without `pt`, and where `\__fp_to_int:w` does more work. To convert to an integer, first round to 0 places (to the nearest integer), then express the result as a decimal number: the definition of `\__fp_to_decimal_dispatch:w` is such that there are no trailing dot nor zero.

```
\cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \__fp_to_int_dispatch:w #1 }
\cs_generate_variant:Nn \fp_to_int:N { c }
\cs_new:Npn \fp_to_int:n
  {
    \exp_after:wN \__fp_to_int_dispatch:w
    \exp:w \exp_end_continue_f:w \__fp_parse:n
  }
\cs_new:Npn \__fp_to_int_dispatch:w #1#2 ;
  {
    \__fp_change_func_type:NNN #1 \__fp_to_int:w \__fp_to_int_recover:w
    #1 #2 ;
  }
\cs_new:Npn \__fp_to_int_recover:w #1
  { \__fp_invalid_operation:nnw { 0 } { fp_to_int } }
\cs_new:Npn \__fp_to_int:w #1;
  {
    \exp_after:wN \__fp_to_decimal:w \exp:w \exp_end_continue_f:w
    \__fp_round:Nwn \__fp_round_to_nearest:NNN #1; { 0 }
  }
```

(*End of definition for* `\fp_to_int:N` *and others. These functions are documented on page 260.*)

78.8 Convert from a dimension

`\dim_to_fp:n`
`\__fp_from_dim_test:ww`
`\__fp_from_dim:wNw`
`\__fp_from_dim:wNNnnnnnn`
`\__fp_from_dim:wnnnnwNw`

The dimension expression (which can in fact be a glue expression) is evaluated, converted to a number (*i.e.*, expressed in scaled points), then multiplied by $2^{-16} = 0.0000152587890625$ to give a value expressed in points. The auxiliary `\__fp_mul_npos_o:Nww` expects the desired ⟨*final sign*⟩ and two floating point operands (of the form `\s__fp` … `;`) as arguments. This set of functions is also used to convert dimension registers to floating points while parsing expressions: in this context there is an additional exponent, which is the first argument of `\__fp_from_dim_test:ww`, and is combined with the exponent -4 of 2^{-16}. There is also a need to expand afterwards: this is performed by `\__fp_mul_npos_o:Nww`, and cancelled by `\prg_do_nothing:` here.

```
\cs_new:Npn \dim_to_fp:n #1
  {
    \exp_after:wN \__fp_from_dim_test:ww
    \exp_after:wN 0
    \exp_after:wN ,
    \int_value:w \tex_glueexpr:D #1 ;
  }
\cs_new:Npn \__fp_from_dim_test:ww #1, #2
  {
    \if_meaning:w 0 #2
      \__fp_case_return:nw { \exp_after:wN \c_zero_fp }
    \else:
      \exp_after:wN \__fp_from_dim:wNw
      \int_value:w \__fp_int_eval:w #1 - 4
        \if_meaning:w - #2
          \exp_after:wN , \exp_after:wN 2 \int_value:w
        \else:
          \exp_after:wN , \exp_after:wN 0 \int_value:w #2
        \fi:
    \fi:
  }
\cs_new:Npn \__fp_from_dim:wNw #1,#2#3;
  {
    \__fp_pack_twice_four:wNNNNNNNN \__fp_from_dim:wNNnnnnnn ;
    #3 000 0000 00 {10}987654321; #2 {#1}
  }
\cs_new:Npn \__fp_from_dim:wNNnnnnnn #1; #2#3#4#5#6#7#8#9
  { \__fp_from_dim:wnnnnwNn #1 {#2#300} {0000} ; }
\cs_new:Npn \__fp_from_dim:wnnnnwNn #1; #2#3#4#5#6; #7#8
  {
    \__fp_mul_npos_o:Nww #7
      \s__fp \__fp_chk:w 1 #7 {#5} #1 ;
      \s__fp \__fp_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} ;
      \prg_do_nothing:
  }
```

(*End of definition for* `\dim_to_fp:n` *and others. This function is documented on page 230.*)

78.9 Use and eval

`\fp_use:N`
`\fp_use:c`
`\fp_eval:n`

Those public functions are simple copies of the decimal conversions.

```
\cs_new_eq:NN \fp_use:N \fp_to_decimal:N
\cs_generate_variant:Nn \fp_use:N { c }
\cs_new_eq:NN \fp_eval:n \fp_to_decimal:n
```

(*End of definition for* `\fp_use:N` *and* `\fp_eval:n`*. These functions are documented on page 261.*)

`\fp_sign:n`

Trivial but useful. See the implementation of `\fp_add:Nn` for an explanation of why to use `\__fp_parse:n`, namely, for better error reporting.

```
\cs_new:Npn \fp_sign:n #1
  { \fp_to_decimal:n { sign \__fp_parse:n {#1} } }
```

(*End of definition for* `\fp_sign:n`. *This function is documented on page 260.*)

`\fp_abs:n` Trivial but useful. See the implementation of `\fp_add:Nn` for an explanation of why to use `\__fp_parse:n`, namely, for better error reporting.

```
\cs_new:Npn \fp_abs:n #1
  { \fp_to_decimal:n { abs \__fp_parse:n {#1} } }
```

(*End of definition for* `\fp_abs:n`. *This function is documented on page 279.*)

`\fp_max:nn`
`\fp_min:nn`
Similar to `\fp_abs:n`, for consistency with `\int_max:nn`, *etc.*

```
\cs_new:Npn \fp_max:nn #1#2
  { \fp_to_decimal:n { max ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
\cs_new:Npn \fp_min:nn #1#2
  { \fp_to_decimal:n { min ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
```

(*End of definition for* `\fp_max:nn` *and* `\fp_min:nn`. *These functions are documented on page 279.*)

78.10 Convert an array of floating points to a comma list

`\__fp_array_to_clist:n`
`\__fp_array_to_clist_loop:Nw`
Converts an array of floating point numbers to a comma-list. If speed here ends up irrelevant, we can simplify the code for the auxiliary to become

```
\cs_new:Npn \__fp_array_to_clist_loop:Nw #1#2;
  {
    \use_none:n #1
    { , ~ } \fp_to_tl:n { #1 #2 ; }
    \__fp_array_to_clist_loop:Nw
  }
```

The `\use_ii:nn` function is expanded after `\__fp_expand:n` is done, and it removes `,~` from the start of the representation.

```
\cs_new:Npn \__fp_array_to_clist:n #1
  {
    \tl_if_empty:nF {#1}
      {
        \exp_last_unbraced:Ne \use_ii:nn
          {
            \__fp_array_to_clist_loop:Nw #1 { ? \prg_break: } ;
            \prg_break_point:
          }
      }
  }
\cs_new:Npn \__fp_array_to_clist_loop:Nw #1#2;
  {
    \use_none:n #1
    , ~
    \exp_not:f { \__fp_to_tl_dispatch:w #1 #2 ; }
    \__fp_array_to_clist_loop:Nw
  }
```

(*End of definition for* `\__fp_array_to_clist:n` *and* `\__fp_array_to_clist_loop:Nw`.)

```
⟨/package⟩
```

Chapter 79

l3fp-random implementation

```
⟨*package⟩
⟨@@=fp⟩
```

__fp_parse_word_rand:N
__fp_parse_word_randint:N

Those functions may receive a variable number of arguments. We won't use the argument ?.

```
\cs_new:Npn \__fp_parse_word_rand:N
  { \__fp_parse_function:NNN \__fp_rand_o:Nw ? }
\cs_new:Npn \__fp_parse_word_randint:N
  { \__fp_parse_function:NNN \__fp_randint_o:Nw ? }
```

(*End of definition for* `\__fp_parse_word_rand:N` *and* `\__fp_parse_word_randint:N`.)

79.1 Engine support

Obviously, every word "random" below means "pseudo-random", as we have no access to entropy (except a very unreliable source of entropy: the time it takes to run some code).

The primitive random number generator (RNG) is provided as `\tex_uniformdeviate:D`. Under the hood, it maintains an array of 55 28-bit numbers, updated with a linear recursion relation (similar to Fibonacci numbers) modulo 2^{28}. When `\tex_uniformdeviate:D` ⟨*integer*⟩ is called (for brevity denote by N the ⟨*integer*⟩), the next 28-bit number is read from the array, scaled by $N/2^{28}$, and rounded. To prevent 0 and N from appearing half as often as other numbers, they are both mapped to the result 0.

This process means that `\tex_uniformdeviate:D` only gives a uniform distribution from 0 to $N-1$ if N is a divisor of 2^{28}, so we will mostly call the RNG with such power of 2 arguments. If N does not divide 2^{28}, then the relative non-uniformity (difference between probabilities of getting different numbers) is about $N/2^{28}$. This implies that detecting deviation from $1/N$ of the probability of a fixed value X requires about $2^{56}/N$ random trials. But collective patterns can reduce this to about $2^{56}/N^2$. For instance with $N = 3\times 2^k$, the modulo 3 repartition of such random numbers is biased with a non-uniformity about $2^k/2^{28}$ (which is much worse than the circa $3/2^{28}$ non-uniformity from taking directly $N = 3$). This is detectable after about $2^{56}/2^{2k} = 9\cdot 2^{56}/N^2$ random numbers. For $k = 15$, $N = 98304$, this means roughly 2^{26} calls to the RNG (experimentally this takes at the very least 16 seconds on a 2 giga-hertz processor). While this bias is not quite problematic, it is uncomfortably close to being so, and it becomes worse as N is increased. In our code, we shall thus combine several results from the RNG.

The RNG has three types of unexpected correlations. First, everything is linear modulo 2^{28}, hence the lowest k bits of the random numbers only depend on the lowest k bits of the seed (and of course the number of times the RNG was called since setting the seed). The recommended way to get a number from 0 to $N-1$ is thus to scale the raw 28-bit integer, as the engine's RNG does. We will go further and in fact typically we discard some of the lowest bits.

Second, suppose that we call the RNG with the same argument N to get a set of K integers in $[0, N-1]$ (throwing away repeats), and suppose that $N > K^3$ and $K > 55$. The recursion used to construct more 28-bit numbers from previous ones is linear: $x_n = x_{n-55} - x_{n-24}$ or $x_n = x_{n-55} - x_{n-24} + 2^{28}$. After rescaling and rounding we find that the result $N_n \in [0, N-1]$ is among $N_{n-55} - N_{n-24} + \{-1, 0, 1\}$ modulo N (a more detailed analysis shows that 0 appears with frequency close to 3/4). The resulting set thus has more triplets (a, b, c) than expected obeying $a = b + c$ modulo N. Namely it will have of order $(K - 55) \times 3/4$ such triplets, when one would expect $K^3/(6N)$. This starts to be detectable around $N = 2^{18} > 55^3$ (earlier if one keeps track of positions too, but this is more subtle than it looks because the array of 28-bit integers is read backwards by the engine). Hopefully the correlation is subtle enough to not affect realistic documents so we do not specifically mitigate against this. Since we typically use two calls to the RNG per `\int_rand:nn` we would need to investigate linear relations between the x_{2n} on the one hand and between the x_{2n+1} on the other hand. Such relations will have more complicated coefficients than ± 1, which alleviates the issue.

Third, consider successive batches of 165 calls to the RNG (with argument 2^{28} or with argument 2 for instance), then most batches have more odd than even numbers. Note that this does not mean that there are more odd than even numbers overall. Similar issues are discussed in Knuth's TAOCP volume 2 near exercise 3.3.2-31. We do not have any mitigation strategy for this.

Ideally, our algorithm should be:

- Uniform. The result should be as uniform as possible assuming that the RNG's underlying 28-bit integers are uniform.
- Uncorrelated. The result should not have detectable correlations between different seeds, similar to the lowest-bit ones mentioned earlier.
- Quick. The algorithm should be fast in TeX, so no "bit twiddling", but "digit twiddling" is ok.
- Simple. The behaviour must be documentable precisely.
- Predictable. The number of calls to the RNG should be the same for any `\int_rand:nn`, because then the algorithm can be modified later without changing the result of other uses of the RNG.
- Robust. It should work even for `\int_rand:nn { - \c_max_int } { \c_max_int }` where the range is not representable as an integer. In fact, we also provide later a floating-point `randint` whose range can go all the way up to $2 \times 10^{16} - 1$ possible values.

Some of these requirements conflict. For instance, uniformity cannot be achieved with a fixed number of calls to the RNG.

Denote by random(N) one call to `\tex_uniformdeviate:D` with argument N, and by ediv(p, q) the ε-TeX rounding division giving $\lfloor p/q + 1/2 \rfloor$. Denote by ⟨*min*⟩, ⟨*max*⟩

and $R = \langle\textit{max}\rangle - \langle\textit{min}\rangle + 1$ the arguments of `\int_min:nn` and the number of possible outcomes. Note that $R \in [1, 2^{32} - 1]$ cannot necessarily be represented as an integer (however, $R - 2^{31}$ can). Our strategy is to get two 28-bit integers X and Y from the RNG, split each into 14-bit integers, as $X = X_1 \times 2^{14} + X_0$ and $Y = Y_1 \times 2^{14} + Y_0$ then return essentially $\langle\textit{min}\rangle + \lfloor R(X_1 \times 2^{-14} + Y_1 \times 2^{-28} + Y_0 \times 2^{-42} + X_0 \times 2^{-56})\rfloor$. For small R the X_0 term has a tiny effect so we ignore it and we can compute $R \times Y/2^{28}$ much more directly by $\mathrm{random}(R)$.

- If $R \leq 2^{17} - 1$ then return $\mathrm{ediv}(R\,\mathrm{random}(2^{14}) + \mathrm{random}(R) + 2^{13}, 2^{14}) - 1 + \langle\textit{min}\rangle$. The shifts by 2^{13} and -1 convert ε-TeX division to truncated division. The bound on R ensures that the number obtained after the shift is less than `\c_max_int`. The non-uniformity is at most of order $2^{17}/2^{42} = 2^{-25}$.

- Split $R = R_2 \times 2^{28} + R_1 \times 2^{14} + R_0$, where $R_2 \in [0, 15]$. Compute $\langle\textit{min}\rangle + R_2 X_1 2^{14} + (R_2 Y_1 + R_1 X_1) + \mathrm{ediv}(R_2 Y_0 + R_1 Y_1 + R_0 X_1 + \mathrm{ediv}(R_2 X_0 + R_0 Y_1 + \mathrm{ediv}((2^{14} R_1 + R_0)(2^{14} Y_0 + X_0), 2^{28}), 2^{14}), 2^{14})$ then map a result of $\langle\textit{max}\rangle + 1$ to $\langle\textit{min}\rangle$. Writing each ediv in terms of truncated division with a shift, and using $\lfloor (p + \lfloor r/s \rfloor)/q \rfloor = \lfloor (ps + r)/(sq) \rfloor$, what we compute is equal to $\lfloor \langle\textit{exact}\rangle + 2^{-29} + 2^{-15} + 2^{-1} \rfloor$ with $\langle\textit{exact}\rangle = \langle\textit{min}\rangle + R \times 0.X_1 Y_1 Y_0 X_0$. Given we map $\langle\textit{max}\rangle + 1$ to $\langle\textit{min}\rangle$, the shift has no effect on uniformity. The non-uniformity is bounded by $R/2^{56} < 2^{-24}$. It may be possible to speed up the code by dropping tiny terms such as $R_0 X_0$, but the analysis of non-uniformity proves too difficult.

  To avoid the overflow when the computation yields $\langle\textit{max}\rangle + 1$ with $\langle\textit{max}\rangle = 2^{31} - 1$ (note that R is then arbitrary), we compute the result in two pieces. Compute $\langle\textit{first}\rangle = \langle\textit{min}\rangle + R_2 X_1 2^{14}$ if $R_2 < 8$ or $\langle\textit{min}\rangle + 8 X_1 2^{14} + (R_2 - 8) X_1 2^{14}$ if $R_2 \geq 8$, the expressions being chosen to avoid overflow. Compute $\langle\textit{second}\rangle = R_2 Y_1 + R_1 X_1 + \mathrm{ediv}(\ldots)$, at most $R_2 2^{14} + R_1 2^{14} + R_0 \leq 2^{28} + 15 \times 2^{14} - 1$, not at risk of overflowing. We have $\langle\textit{first}\rangle + \langle\textit{second}\rangle = \langle\textit{max}\rangle + 1 = \langle\textit{min}\rangle + R$ if and only if $\langle\textit{second}\rangle = R1 2^{14} + R_0 + R_2 2^{14}$ and $2^{14} R_2 X_1 = 2^{28} R_2 - 2^{14} R_2$ (namely $R_2 = 0$ or $X_1 = 2^{14} - 1$). In that case, return $\langle\textit{min}\rangle$, otherwise return $\langle\textit{first}\rangle + \langle\textit{second}\rangle$, which is safe because it is at most $\langle\textit{max}\rangle$. Note that the decision of what to return does not need $\langle\textit{first}\rangle$ explicitly so we don't actually compute it, just put it in an integer expression in which $\langle\textit{second}\rangle$ is eventually added (or not).

- To get a floating point number in $[0, 1)$ just call the $R = 10000 \leq 2^{17} - 1$ procedure above to produce four blocks of four digits.

- To get an integer floating point number in a range (whose size can be up to $2 \times 10^{16} - 1$), work with fixed-point numbers: get six times four digits to build a fixed point number, multiply by R and add $\langle\textit{min}\rangle$. This requires some care because l3fp-extended only supports non-negative numbers.

`\c__kernel_randint_max_int` Constant equal to $2^{17} - 1$, the maximal size of a range that `\int_range:nn` can do with its "simple" algorithm.

```
\int_const:Nn \c__kernel_randint_max_int { 131071 }
```

(*End of definition for* `\c__kernel_randint_max_int`*.*)

`\__kernel_randint:n` Used in an integer expression, `\__kernel_randint:n` $\{R\}$ gives a random number $1 + \lfloor (R\,\mathrm{random}(2^{14}) + \mathrm{random}(R))/2^{14} \rfloor$ that is in $[1, R]$. Previous code was computing $\lfloor p/2^{14} \rfloor$ as $\mathrm{ediv}(p - 2^{13}, 2^{14})$ but that wrongly gives -1 for $p = 0$.

```
\cs_new:Npn \__kernel_randint:n #1
  {
    (#1 * \tex_uniformdeviate:D 16384
    + \tex_uniformdeviate:D #1 + 8192 ) / 16384
  }
```

(*End of definition for* `\__kernel_randint:n`.)

`\__fp_rand_myriads:n`
`\__fp_rand_myriads_loop:w`
`\__fp_rand_myriads_get:w`

Used as `\__fp_rand_myriads:n {XXX}` with one letter `X` (specifically) per block of four digit we want; it expands to `;` followed by the requested number of brace groups, each containing four (pseudo-random) digits. Digits are produced as a random number in $[10000, 19999]$ for the usual reason of preserving leading zeros.

```
\cs_new:Npn \__fp_rand_myriads:n #1
  { \__fp_rand_myriads_loop:w #1 \prg_break: X \prg_break_point: ; }
\cs_new:Npn \__fp_rand_myriads_loop:w #1 X
  {
    #1
    \exp_after:wN \__fp_rand_myriads_get:w
    \int_value:w \__fp_int_eval:w 9999 +
      \__kernel_randint:n { 10000 }
    \__fp_rand_myriads_loop:w
  }
\cs_new:Npn \__fp_rand_myriads_get:w 1 #1 ; { ; {#1} }
```

(*End of definition for* `\__fp_rand_myriads:n`, `\__fp_rand_myriads_loop:w`, *and* `\__fp_rand_myriads_get:w`.)

79.2 Random floating point

`\__fp_rand_o:Nw`
`\__fp_rand_o:w`

First we check that `random` was called without argument. Then get four blocks of four digits and convert that fixed point number to a floating point number (this correctly sets the exponent). This has a minor bug: if all of the random numbers are zero then the result is correctly 0 but it raises the `underflow` flag; it should not do that.

```
\cs_new:Npn \__fp_rand_o:Nw ? #1 @
  {
    \tl_if_empty:nTF {#1}
      {
        \exp_after:wN \__fp_rand_o:w
        \exp:w \exp_end_continue_f:w
        \__fp_rand_myriads:n { XXXX } { 0000 } { 0000 } ; 0
      }
      {
        \msg_expandable_error:nnnnn
          { fp } { num-args } { rand() } { 0 } { 0 }
        \exp_after:wN \c_nan_fp
      }
  }
\cs_new:Npn \__fp_rand_o:w ;
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN 0
    \int_value:w \__fp_int_eval:w \c_zero_int
      \__fp_fixed_to_float_o:wN
  }
```

(*End of definition for* `\__fp_rand_o:Nw` *and* `\__fp_rand_o:w`.)

79.3 Random integer

`\__fp_randint_o:Nw`
`\__fp_randint_default:w`
`\__fp_randint_badarg:w`
`\__fp_randint_o:w`
`\__fp_randint_auxi_o:ww`
`\__fp_randint_auxii:wn`
`\__fp_randint_auxiii_o:ww`
`\__fp_randint_auxiv_o:ww`
`\__fp_randint_auxv_o:w`

Enforce that there is one argument (then add first argument 1) or two arguments. Call `\__fp_randint_badarg:w` on each; this function inserts `1 \exp_stop_f:` to end the `\if_case:w` statement if either the argument is not an integer or if its absolute value is $\geq 10^{16}$. Also bail out if `\__fp_compare_back:ww` yields `1`, meaning that the bounds are not in the right order. Otherwise an auxiliary converts each argument times 10^{-16} (hence the shift in exponent) to a 24-digit fixed point number (see l3fp-extended). Then compute the number of choices, $\langle max\rangle + 1 - \langle min\rangle$. Create a random 24-digit fixed-point number with `\__fp_rand_myriads:n`, then use a fused multiply-add instruction to multiply the number of choices to that random number and add it to $\langle min\rangle$. Then truncate to 16 digits (namely select the integer part of 10^{16} times the result) before converting back to a floating point number (`\__fp_sanitize:Nw` takes care of zero). To avoid issues with negative numbers, add 1 to all fixed point numbers (namely 10^{16} to the integers they represent), except of course when it is time to convert back to a float.

```
\cs_new:Npn \__fp_randint_o:Nw ?
  {
    \__fp_parse_function_one_two:nnw
      { randint }
      { \__fp_randint_default:w \__fp_randint_o:w }
  }
\cs_new:Npn \__fp_randint_default:w #1 { \exp_after:wN #1 \c_one_fp }
\cs_new:Npn \__fp_randint_badarg:w \s__fp \__fp_chk:w #1#2#3;
  {
    \__fp_int:wTF \s__fp \__fp_chk:w #1#2#3;
      {
        \if_meaning:w 1 #1
          \if_int_compare:w
              \__fp_use_i_until_s:nw #3 ; > \c__fp_prec_int
            \c_one_int
          \fi:
        \fi:
      }
      { \c_one_int }
  }
\cs_new:Npn \__fp_randint_o:w #1; #2; @
  {
    \if_case:w
        \__fp_randint_badarg:w #1;
        \__fp_randint_badarg:w #2;
        \if:w 1 \__fp_compare_back:ww #2; #1; \c_one_int \fi:
        \c_zero_int
      \__fp_randint_auxi_o:ww #1; #2;
    \or:
      \__fp_invalid_operation_tl_o:ff
        { randint } { \__fp_array_to_clist:n { #1; #2; } }
      \exp:w
    \fi:
    \exp_after:wN \exp_end:
  }
```

```
\cs_new:Npn \__fp_randint_auxi_o:ww #1 ; #2 ; #3 \exp_end:
  {
    \fi:
    \__fp_randint_auxii:wn #2 ;
    { \__fp_randint_auxii:wn #1 ; \__fp_randint_auxiii_o:ww }
  }
\cs_new:Npn \__fp_randint_auxii:wn \s__fp \__fp_chk:w #1#2#3#4 ;
  {
    \if_meaning:w 0 #1
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
    { \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl }
    {
      \exp_after:wN \__fp_ep_to_fixed:wwn
      \int_value:w \__fp_int_eval:w
        #3 - \c__fp_prec_int , #4 {0000} {0000} ;
      {
        \if_meaning:w 0 #2
          \exp_after:wN \use_i:nnnn
          \exp_after:wN \__fp_fixed_add_one:wN
        \fi:
        \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl
      }
      \__fp_fixed_continue:wn
    }
  }
\cs_new:Npn \__fp_randint_auxiii_o:ww #1 ; #2 ;
  {
    \__fp_fixed_add:wwn #2 ;
      {0000} {0000} {0000} {0001} {0000} {0000} ;
    \__fp_fixed_sub:wwn #1 ;
    {
      \exp_after:wN \use_i:nn
      \exp_after:wN \__fp_fixed_mul_add:wwwn
      \exp:w \exp_end_continue_f:w \__fp_rand_myriads:n { XXXXXX } ;
    }
    #1 ;
    \__fp_randint_auxiv_o:ww
    #2 ;
    \__fp_randint_auxv_o:w #1 ; @
  }
\cs_new:Npn \__fp_randint_auxiv_o:ww #1#2#3#4#5 ; #6#7#8#9
  {
    \if_int_compare:w
      \if_int_compare:w #1#2 > #6#7 \exp_stop_f: 1 \else:
        \if_int_compare:w #1#2 < #6#7 \exp_stop_f: - \fi: \fi:
      #3#4 > #8#9 \exp_stop_f:
      \__fp_use_i_until_s:nw
    \fi:
    \__fp_randint_auxv_o:w {#1}{#2}{#3}{#4}#5
  }
\cs_new:Npn \__fp_randint_auxv_o:w #1#2#3#4#5 ; #6 @
```

```
  {
    \exp_after:wN \__fp_sanitize:Nw
    \int_value:w
    \if_int_compare:w #1 < 10000 \exp_stop_f:
      2
    \else:
      0
      \exp_after:wN \exp_after:wN
      \exp_after:wN \__fp_reverse_args:Nww
    \fi:
    \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl
    {#1} {#2} {#3} {#4} {0000} {0000} ;
    {
      \exp_after:wN \exp_stop_f:
      \int_value:w \__fp_int_eval:w \c__fp_prec_int
        \__fp_fixed_to_float_o:wN
    }
    0
    \exp:w \exp_after:wN \exp_end:
  }
```

(*End of definition for* `\__fp_randint_o:Nw` *and others.*)

`\int_rand:nn`
`\__fp_randint:ww`

Evaluate the argument and filter out the case where the lower bound `#1` is more than the upper bound `#2`. Then determine whether the range is narrower than `\c__kernel_randint_max_int`; `#2-#1` may overflow for very large positive `#2` and negative `#1`. If the range is narrow, call `\__kernel_randint:n` {⟨*choices*⟩} where ⟨`choices`⟩ is the number of possible outcomes. If the range is wide, use somewhat slower code.

```
\cs_new:Npn \int_rand:nn #1#2
  {
    \int_eval:n
      {
        \exp_after:wN \__fp_randint:ww
        \int_value:w \int_eval:n {#1} \exp_after:wN ;
        \int_value:w \int_eval:n {#2} ;
      }
  }
\cs_new:Npn \__fp_randint:ww #1; #2;
  {
    \if_int_compare:w #1 > #2 \exp_stop_f:
      \msg_expandable_error:nnnn
        { kernel } { randint-backward-range } {#1} {#2}
      \__fp_randint:ww #2; #1;
    \else:
      \if_int_compare:w \__fp_int_eval:w #2
          \if_int_compare:w #1 > \c_zero_int
            - #1 < \__fp_int_eval:w
          \else:
            < \__fp_int_eval:w #1 +
          \fi:
          \c__kernel_randint_max_int
          \__fp_int_eval_end:
        \__kernel_randint:n
          { \__fp_int_eval:w #2 - #1 + 1 \__fp_int_eval_end: }
```

```
      - 1 + #1
    \else:
      \__kernel_randint:nn {#1} {#2}
    \fi:
  \fi:
}
```

(*End of definition for* `\int_rand:nn` *and* `\__fp_randint:ww`*. This function is documented on page 176.*)

`\__kernel_randint:nn`
`\__fp_randint_split_o:Nw`
`\__fp_randint_split_aux:w`
`\__fp_randinat_wide_aux:w`
`\__fp_randinat_wide_auxii:w`

Any $n \in [-2^{31}+1, 2^{31}-1]$ is uniquely written as $2^{14}n_1 + n_2$ with $n_1 \in [-2^{17}, 2^{17}-1]$ and $n_2 \in [0, 2^{14}-1]$. Calling `\__fp_randint_split_o:Nw` n `;` gives n_1`;` n_2`;` and expands the next token once. We do this for two random numbers and apply `\__fp_randint_split_o:Nw` twice to fully decompose the range R. One subtlety is that we compute $R - 2^{31} = \langle max\rangle - \langle min\rangle - (2^{31}-1) \in [-2^{31}+1, 2^{31}-1]$ rather than R to avoid overflow.

Then we have `\__fp_randint_wide_aux:w` $\langle X_1\rangle$`;`$\langle X_0\rangle$`;` $\langle Y_1\rangle$`;`$\langle Y_0\rangle$`;` $\langle R_2\rangle$`;`$\langle R_1\rangle$`;`$\langle R_0\rangle$`;.` and we apply the algorithm described earlier.

```
\cs_new:Npn \__kernel_randint:nn #1#2
  {
    #1
    \exp_after:wN \__fp_randint_wide_aux:w
    \int_value:w
      \exp_after:wN \__fp_randint_split_o:Nw
      \tex_uniformdeviate:D 268435456 ;
    \int_value:w
      \exp_after:wN \__fp_randint_split_o:Nw
      \tex_uniformdeviate:D 268435456 ;
    \int_value:w
      \exp_after:wN \__fp_randint_split_o:Nw
      \int_value:w \__fp_int_eval:w 131072 +
        \exp_after:wN \__fp_randint_split_o:Nw
        \int_value:w
          \__kernel_int_add:nnn {#2} { -#1 } { -\c_max_int } ;
    .
  }
\cs_new:Npn \__fp_randint_split_o:Nw #1#2 ;
  {
    \if_meaning:w 0 #1
      0 \exp_after:wN ; \int_value:w 0
    \else:
      \exp_after:wN \__fp_randint_split_aux:w
      \int_value:w \__fp_int_eval:w (#1#2 - 8192) / 16384 ;
      + #1#2
    \fi:
    \exp_after:wN ;
  }
\cs_new:Npn \__fp_randint_split_aux:w #1 ;
  {
    #1 \exp_after:wN ;
    \int_value:w \__fp_int_eval:w - #1 * 16384
  }
\cs_new:Npn \__fp_randint_wide_aux:w #1;#2; #3;#4; #5;#6;#7; .
  {
    \exp_after:wN \__fp_randint_wide_auxii:w
    \int_value:w \__fp_int_eval:w #5 * #3 + #6 * #1 +
```

```
      (#5 * #4 + #6 * #3 + #7 * #1 +
        (#5 * #2 +           #7 * #3 +
          (16384 * #6 + #7) * (16384 * #4 + #2) / 268435456) / 16384
      ) / 16384 \exp_after:wN ;
    \int_value:w \__fp_int_eval:w (#5 + #6) * 16384 + #7 ;
    #1 ; #5 ;
  }
\cs_new:Npn \__fp_randint_wide_auxii:w #1; #2; #3; #4;
  {
    \if_int_odd:w 0
        \if_int_compare:w #1 = #2 \else: \exp_stop_f: \fi:
        \if_int_compare:w #4 = \c_zero_int 1 \fi:
        \if_int_compare:w #3 = 16383 ~ 1 \fi:
        \exp_stop_f:
      \exp_after:wN \prg_break:
    \fi:
    \if_int_compare:w #4 < 8 \exp_stop_f:
      + #4 * #3 * 16384
    \else:
      + 8 * #3 * 16384 + (#4 - 8) * #3 * 16384
    \fi:
    + #1
    \prg_break_point:
  }
```

(*End of definition for* `\__kernel_randint:nn` *and others.*)

`\int_rand:n` `\__fp_randint:n`

Similar to `\int_rand:nn`, but needs fewer checks.

```
\cs_new:Npn \int_rand:n #1
  {
    \int_eval:n
      { \exp_args:Nf \__fp_randint:n { \int_eval:n {#1} } }
  }
\cs_new:Npn \__fp_randint:n #1
  {
    \if_int_compare:w #1 < \c_one_int
      \msg_expandable_error:nnnn
        { kernel } { randint-backward-range } { 1 } {#1}
      \__fp_randint:ww #1; 1;
    \else:
      \if_int_compare:w #1 > \c__kernel_randint_max_int
        \__kernel_randint:nn { 1 } {#1}
      \else:
        \__kernel_randint:n {#1}
      \fi:
    \fi:
  }
```

(*End of definition for* `\int_rand:n` *and* `\__fp_randint:n`*. This function is documented on page 176.*)

```
⟨/package⟩
```

Chapter 80

l3fp-types implementation

```
⟨*package⟩
⟨@@=fp⟩
```

80.1 Support for types

Despite lack of documentation, the l3fp internals support types. Each additional type must define

- `\s__fp_`⟨*type*⟩ and `\__fp_`⟨*type*⟩`_chk:w`;
- `\__fp_exp_after_`⟨*type*⟩`_f:nw`;
- `\__fp_`⟨*type*⟩`_to_`⟨*out*⟩`:w` for ⟨*out*⟩ among `decimal`, `scientific`, `tl`;

and may define

- `\__fp_`⟨*type*⟩`_to_int:w` and `\__fp_`⟨*type*⟩`_to_dim:w`;
- `\__fp_`⟨*op*⟩`_`⟨*type*⟩`_o:w` for any of the ⟨*op*⟩ that the type implements, among `acos`, `acsc`, `asec`, `asin`, `cos`, `cot`, `csc`, `exp`, `ln`, `not`, `sec`, `set_sign`, `sin`, `tan`;
- `\__fp_`⟨*type*$_1$⟩`_`⟨*op*⟩`_`⟨*type*$_2$⟩`_o:ww` for ⟨*op*⟩ among `^*/-+&|` and for every pair of types;
- `\__fp_`⟨*type*$_1$⟩`_bcmp_`⟨*type*$_2$⟩`:ww` for every pair of types.

The latter is set up in l3fp-logic.

80.2 Dispatch according to the type

`\__fp_types_cs_to_op:N`
`\__fp_types_cs_to_op_auxi:wwwn`

From `\__fp_`⟨*op*⟩`_o:w` produce ⟨*op*⟩, otherwise `?`.

```
\cs_new:Npe \__fp_types_cs_to_op:N #1
  {
    \exp_not:N \exp_after:wN \exp_not:N \__fp_types_cs_to_op_auxi:wwwn
      \exp_not:N \token_to_str:N #1 \s__fp_mark
      \exp_not:N \__fp_use_i_delimit_by_s_stop:nw
      \tl_to_str:n { __fp_ _o:w } \s__fp_mark
        { \exp_not:N \__fp_use_i_delimit_by_s_stop:nw ? }
```

```
      \s__fp_stop
  }
\use:e
  {
    \cs_new:Npn \exp_not:N \__fp_types_cs_to_op_auxi:wwwn
      #1 \tl_to_str:n { __fp_ } #2
      \tl_to_str:n { _o:w } #3 \s__fp_mark #4 { #4 {#2} }
  }
```

(*End of definition for* `\__fp_types_cs_to_op:N` *and* `\__fp_types_cs_to_op_auxi:wwwn`.)

`\__fp_types_unary:NNw`
`\__fp_types_unary_auxi:nNw`
`\__fp_types_unary_auxii:NnNw`

`\__fp_types_unary:NNw \__fp_⟨function⟩_o:w ⟨token⟩ ⟨operand⟩ @`

```
\cs_new:Npn \__fp_types_unary:NNw #1
  {
    \exp_args:Nf \__fp_types_unary_auxi:nNw
      { \__fp_types_cs_to_op:N #1 }
  }
\cs_new:Npn \__fp_types_unary_auxi:nNw #1#2#3
  {
    \exp_after:wN \__fp_types_unary_auxii:NnNw
    \cs:w __fp_#1 \__fp_type_from_scan:N #3 _o:w \cs_end:
    {#1}
    #2#3
  }
\cs_new:Npn \__fp_types_unary_auxii:NnNw #1#2#3
  {
    \token_if_eq_meaning:NNTF \scan_stop: #1
      { \__fp_invalid_operation_o:nw {#2} }
      { #1 #3 }
  }
```

(*End of definition for* `\__fp_types_unary:NNw`, `\__fp_types_unary_auxi:nNw`, *and* `\__fp_types_unary_auxii:NnNw`.)

`\__fp_types_binary:Nww`
`\__fp_types_binary_auxi:Nww`
`\__fp_types_binary_auxii:NNww`

`\__fp_types_binary:Nww \__fp_⟨binop⟩_o:ww ⟨operand₁⟩ ⟨operand₂⟩ @`

```
\cs_new:Npn \__fp_types_binary:Nww #1
  {
    \exp_last_unbraced:Nf \__fp_types_binary_auxi:Nww
      { \__fp_types_cs_to_op:N #1 }
  }
\cs_new:Npn \__fp_types_binary_auxi:Nww #1#2#3; #4#5; @
  {
    \exp_after:wN \__fp_types_binary_auxii:NNww
    \cs:w
      __fp
      \__fp_type_from_scan:N #2
      _#1
      \__fp_type_from_scan:N #4
      _o:ww
    \cs_end:
    #1 #2#3; #4#5;
  }
```

```
\cs_new:Npn \__fp_types_binary_auxii:NNww #1#2
  {
    \token_if_eq_meaning:NNTF \scan_stop: #1
      { \__fp_invalid_operation_o:Nww #2 }
      {#1}
  }
```

(*End of definition for* `\__fp_types_binary:Nww`*,* `\__fp_types_binary_auxi:Nww`*, and* `\__fp_types_binary_auxii:NNww`*.*)

```
⟨/package⟩
```

Chapter 81

l3fp-symbolic implementation

```
⟨*package⟩
⟨@@=fp⟩
```

81.1 Misc

\l__fp_symbolic_fp Scratch floating point.

```
\fp_new:N \l__fp_symbolic_fp
```

(*End of definition for* `\l__fp_symbolic_fp`*.*)

81.2 Building blocks for expressions

Every symbolic expression has the form `\s__fp_symbolic \__fp_symbolic_chk:w` ⟨*operation*⟩ `,` `{`⟨*operands*⟩`}` ⟨*junk*⟩ `;` where the ⟨*operation*⟩ is a list of `N`-type tokens, the ⟨*operands*⟩ is an array of floating point objects, and the ⟨*junk*⟩ is to be discarded. If the outermost operator (last to be evaluated) is unary, the expression has the form

```
\s__fp_symbolic \__fp_symbolic_chk:w
\__fp_types_unary:NNw \__fp_⟨op⟩_o:w ⟨token⟩ ,
{ ⟨operand⟩ } ⟨junk⟩ ;
```

where the ⟨*op*⟩ is a unary operation (`set_sign`, `cos`, …), and the ⟨*token*⟩ and ⟨*operand*⟩ are used as arguments for `\__fp_`⟨*op*⟩`_o:w` (or the type-specific analog of this function). If the outermost operator is binary, the expression has the form

```
\s__fp_symbolic \__fp_symbolic_chk:w
\__fp_types_binary:Nww \__fp_⟨op⟩_o:ww ,
{ ⟨operand1⟩ ⟨operand2⟩ } ⟨junk⟩ ;
```

where the ⟨*op*⟩ is an operation (`+`, `&`, …), and `\__fp_`⟨*op*⟩`_o:ww` receives the ⟨*operands*⟩ as arguments. If the expression consists of a single variable, it is stored as

```
\s__fp_symbolic \__fp_symbolic_chk:w
\__fp_variable_o:w ⟨identifier⟩ ,
{ } ⟨junk⟩ ;
```

Symbolic expressions are stored in a prefix form. When encountering a symbolic expression in a floating point computation, we attempt to evaluate the operands as much as possible, and if that yields floating point numbers rather than expressions, we apply the operator which follows (if the function is known).

For instance, the expression `a + b * sin(c)` is stored as

```
\s__fp_symbolic \__fp_symbolic_chk:w
  \__fp_types_binary:Nww \__fp_+_o:ww ,
  {
    \s__fp_symbolic \__fp_symbolic_chk:w
      \__fp_variable_o:w a , { } ;
    \s__fp_symbolic \__fp_symbolic_chk:w
      \__fp_types_binary:Nww \__fp_*_o:ww ,
      {
        \s__fp_symbolic \__fp_symbolic_chk:w
          \__fp_variable_o:w b , { } ;
        \s__fp_symbolic \__fp_symbolic_chk:w
          \__fp_types_unary:NNw \__fp_sin_o:w \use_i:nn ,
          {
            \s__fp_symbolic \__fp_symbolic_chk:w
              \__fp_variable_o:w c , { } ;
          } ;
      } ;
  } ;
```

`\s__fp_symbolic` Scan mark indicating the start of a symbolic expression.

```
\scan_new:N \s__fp_symbolic
```

(*End of definition for* `\s__fp_symbolic`.)

`\__fp_symbolic_chk:w` Analog of `\__fp_chk:w` for symbolic expressions.

```
\cs_new_protected:Npn \__fp_symbolic_chk:w #1,#2#3;
  {
    \msg_error:nne { fp } { misused-fp }
      {
        \__fp_to_tl_dispatch:w
          \s__fp_symbolic \__fp_symbolic_chk:w #1,{#2};
      }
  }
```

(*End of definition for* `\__fp_symbolic_chk:w`.)

81.3 Expanding after a symbolic expression

`\__fp_if_has_symbolic:nTF`
`\__fp_if_has_symbolic_aux:w`

Tests if `#1` contains `\s__fp_symbolic` at top-level. This test should be precise enough to determine if a given array contains a symbolic expression or only consists of floating points. Used in `\__fp_exp_after_symbolic_f:nw`.

```
\cs_new:Npn \__fp_if_has_symbolic:nTF #1
  {
    \__fp_if_has_symbolic_aux:w
      #1             \s__fp_mark \use_i:nn
```

```
      \s__fp_symbolic \s__fp_mark \use_ii:nn
    \s__fp_stop
  }
\cs_new:Npn \__fp_if_has_symbolic_aux:w
    #1 \s__fp_symbolic #2 \s__fp_mark #3#4 \s__fp_stop { #3 }
```

(*End of definition for* `\__fp_if_has_symbolic:nTF` *and* `\__fp_if_has_symbolic_aux:w`.)

`\__fp_exp_after_symbolic_f:nw`
`\__fp_exp_after_symbolic_aux:w`
`\__fp_exp_after_symbolic_loop:N`

This function does two things: trigger an `f`-expansion of the argument `#1` after the following symbolic expression, and evaluate all pieces of the expression which can be evaluated.

```
\cs_new:Npn \__fp_exp_after_symbolic_f:nw
    #1 \s__fp_symbolic \__fp_symbolic_chk:w #2, #3#4;
  {
    \exp_after:wN \__fp_exp_after_symbolic_aux:w
    \exp:w
    \__fp_exp_after_symbolic_loop:N #2
      { , \exp:w \use_none:nn }
    \exp_after:wN \exp_end: \exp_after:wN
      {
        \exp:w \exp_end_continue_f:w
        \__fp_exp_after_array_f:w #3 \s__fp_expr_stop
        \exp_after:wN
      }
    \exp_after:wN ;
    \exp:w \exp_end_continue_f:w #1
  }
\cs_new:Npn \__fp_exp_after_symbolic_aux:w #1, #2;
  {
    \__fp_if_has_symbolic:nTF {#2}
      { \s__fp_symbolic \__fp_symbolic_chk:w #1, {#2} ; }
      { #1 #2 @ \prg_do_nothing: }
  }
\cs_new:Npn \__fp_exp_after_symbolic_loop:N #1
  {
    \exp_after:wN \exp_end:
    \exp_after:wN #1
    \exp:w
    \__fp_exp_after_symbolic_loop:N
  }
```

(*End of definition for* `\__fp_exp_after_symbolic_f:nw`, `\__fp_exp_after_symbolic_aux:w`, *and* `\__fp_exp_after_symbolic_loop:N`.)

81.4 Applying infix operators to expressions

`\__fp_symbolic_binary_o:Nww`

Used when applying infix operators to expressions.

```
\cs_new:Npn \__fp_symbolic_binary_o:Nww #1 #2; #3;
  {
    \__fp_exp_after_symbolic_f:nw { \exp_after:wN \exp_stop_f: }
      \s__fp_symbolic \__fp_symbolic_chk:w
      \__fp_types_binary:Nww #1 , { #2; #3; } ;
  }
```

(*End of definition for* `\__fp_symbolic_binary_o:Nww`.)
^^A Hack! ^^A Hack! ^^A Hack!

`\__fp_symbolic_+_symbolic_o:ww`
`\__fp_symbolic_+_o:ww`
`\__fp_+_symbolic_o:ww`
`\__fp_symbolic_-_symbolic_o:ww`
`\__fp_symbolic_-_o:ww`
`\__fp_-_symbolic_o:ww`
`\__fp_symbolic_*_symbolic_o:ww`
`\__fp_symbolic_*_o:ww`
`\__fp_*_symbolic_o:ww`
`\__fp_symbolic_/_symbolic_o:ww`
`\__fp_symbolic_/_o:ww`
`\__fp_/_symbolic_o:ww`
`\__fp_symbolic_^_symbolic_o:ww`
`\__fp_symbolic_^_o:ww`
`\__fp_^_symbolic_o:ww`
`\__fp_symbolic_|_symbolic_o:ww`
`\__fp_symbolic_|_o:ww`
`\__fp_|_symbolic_o:ww`
`\__fp_symbolic_&_symbolic_o:ww`
`\__fp_symbolic_&_o:ww`
`\__fp_&_symbolic_o:ww`

```
\cs_set_protected:Npn \__fp_tmp:w #1#2
  {
    \cs_new:cpn
      { __fp_symbolic_#2_symbolic_o:ww }
      { \__fp_symbolic_binary_o:Nww #1 }
    \cs_new_eq:cc
      { __fp_symbolic_#2         _o:ww }
      { __fp_symbolic_#2_symbolic_o:ww }
    \cs_new_eq:cc
      { __fp         _#2_symbolic_o:ww }
      { __fp_symbolic_#2_symbolic_o:ww }
  }
\tl_map_inline:nn { + - * / ^ & | }
  { \exp_args:Nc \__fp_tmp:w { __fp_#1_o:ww } {#1} }
```

(*End of definition for* `\__fp_symbolic_+_symbolic_o:ww` *and others.*)

81.5 Applying prefix functions to expressions

`\__fp_symbolic_unary_o:NNw`

Used when applying infix operators to expressions.

```
\cs_new:Npn \__fp_symbolic_unary_o:NNw #1#2#3; @
  {
    \__fp_exp_after_symbolic_f:nw { \exp_after:wN \exp_stop_f: }
      \s__fp_symbolic \__fp_symbolic_chk:w
      \__fp_types_unary:NNw #1#2 , { #3; } ;
  }
```

(*End of definition for* `\__fp_symbolic_unary_o:NNw`.)

`\__fp_symbolic_acos_o:w`
`\__fp_symbolic_acsc_o:w`
`\__fp_symbolic_asec_o:w`
`\__fp_symbolic_asin_o:w`
`\__fp_symbolic_cos_o:w`
`\__fp_symbolic_cot_o:w`
`\__fp_symbolic_csc_o:w`
`\__fp_symbolic_exp_o:w`
`\__fp_symbolic_ln_o:w`
`\__fp_symbolic_not_o:w`
`\__fp_symbolic_sec_o:w`
`\__fp_symbolic_set_sign_o:w`
`\__fp_symbolic_sin_o:w`
`\__fp_symbolic_tan_o:w`

```
\tl_map_inline:nn
  {
    {acos} {acsc} {asec} {asin} {cos} {cot} {csc} {exp} {ln}
    {not} {sec} {set_sign} {sin} {sqrt} {tan}
  }
  {
    \cs_new:cpe { __fp_symbolic_#1_o:w }
      {
        \exp_not:N \__fp_symbolic_unary_o:NNw
        \exp_not:c { __fp_#1_o:w }
      }
  }
```

(*End of definition for* `\__fp_symbolic_acos_o:w` *and others.*)

81.6 Conversions

__fp_symbolic_to_decimal:w
__fp_symbolic_to_int:w
__fp_symbolic_to_scientific:w
__fp_symbolic_convert:wnnN

Symbolic expressions cannot be converted to decimal, integer, or scientific notation unless they can be reduced to

```
\cs_set_protected:Npn \__fp_tmp:w #1#2#3
  {
    \cs_new:cpn { __fp_symbolic_to_#1:w }
      {
        \exp_after:wN \__fp_symbolic_convert:wnnN
        \exp:w \exp_end_continue_f:w
        \__fp_exp_after_symbolic_f:nw { { #2 } { fp_to_#1 } #3 }
      }
  }
\__fp_tmp:w { decimal    } { 0   } \__fp_to_decimal_dispatch:w
\__fp_tmp:w { int        } { 0   } \__fp_to_int_dispatch:w
\__fp_tmp:w { scientific } { nan } \__fp_to_scientific_dispatch:w
\cs_new:Npn \__fp_symbolic_convert:wnnN #1#2; #3#4#5
  {
    \str_if_eq:nnTF {#1} { \s__fp_symbolic }
      { \__fp_invalid_operation:nnw {#3} {#4} #1#2; }
      { #5 #1#2; }
  }
```

(*End of definition for* `\__fp_symbolic_to_decimal:w` *and others.*)

__fp_symbolic_cs_arg_to_fn:NN
__fp_symbolic_op_arg_to_fn:nN

```
\cs_new:Npn \__fp_symbolic_cs_arg_to_fn:NN #1
  {
    \exp_args:Nf \__fp_symbolic_op_arg_to_fn:nN
      { \__fp_types_cs_to_op:N #1 }
  }
\cs_new:Npn \__fp_symbolic_op_arg_to_fn:nN #1#2
  {
    \str_case:nnF { #1 #2 }
      {
        { not ? } { ! }
        { set_sign 0 } { abs }
        { set_sign 2 } { - }
      }
      {
        \token_if_eq_meaning:NNTF #2 \use_ii:nn
          { #1 d } {#1}
      }
  }
```

(*End of definition for* `\__fp_symbolic_cs_arg_to_fn:NN` *and* `\__fp_symbolic_op_arg_to_fn:nN`*.*)

__fp_symbolic_to_tl:w
__fp_symbolic_unary_to_tl:NNw
__fp_symbolic_binary_to_tl:Nww
__fp_symbolic_function_to_tl:Nw

Converting a symbolic expression to a token list is possible.

```
\cs_new:Npn \__fp_symbolic_to_tl:w
    \s__fp_symbolic \__fp_symbolic_chk:w #1#2, #3#4;
  {
    \str_case:nnTF {#1}
      {
        { \__fp_types_unary:NNw } { \__fp_symbolic_unary_to_tl:NNw }
```

```
        { \__fp_types_binary:Nww } { \__fp_symbolic_binary_to_tl:Nww }
        { \__fp_function_o:w } { \__fp_symbolic_function_to_tl:Nw }
      }
      { #2, #3 @ }
      { \tl_to_str:n {#2} }
  }
\cs_new:Npn \__fp_symbolic_unary_to_tl:NNw #1#2 , #3 @
  {
    \use:e
      {
        \__fp_symbolic_cs_arg_to_fn:NN #1#2
        ( \__fp_to_tl_dispatch:w #3 )
      }
  }
\cs_new:Npn \__fp_symbolic_binary_to_tl:Nww #1, #2; #3; @
  {
    \use:e
      {
        ( \__fp_to_tl_dispatch:w #2; )
        \__fp_types_cs_to_op:N #1
        ( \__fp_to_tl_dispatch:w #3; )
      }
  }
\cs_new:Npn \__fp_symbolic_function_to_tl:Nw #1, #2@
  {
    \use:e
      {
        \__fp_types_cs_to_op:N #1
        ( \__fp_array_to_clist:n {#2} )
      }
  }
```

(*End of definition for* `\__fp_symbolic_to_tl:w` *and others.*)

81.7 Identifiers

Functions defined here are not necessarily tied to symbolic expressions.

`\__fp_id_if_invalid:nTF`
`\__fp_id_if_invalid_aux:N`

If `#1` contains a space, it is not a valid identifier. Otherwise, loop through letters in `#1`: if it is not a letter, break the loop and return `true`. If the end of the loop is reached without finding any non-letter, return `false`. Note `#1` must be a str (i.e., resulted from `\tl_to_str:n`).

```
\prg_new_protected_conditional:Npnn
    \__fp_id_if_invalid:n #1 { T , F , TF }
  {
    \tl_if_empty:nTF {#1}
      { \prg_return_true: }
      {
        \tl_if_in:nnTF { #1 } { ~ }
          { \prg_return_true: }
          {
            \__fp_id_if_invalid_aux:N #1
              { ? \prg_break:n \prg_return_false: }
```

```
          \prg_break_point:
        }
    }
  }
\cs_new:Npn \__fp_id_if_invalid_aux:N #1
  {
    \use_none:n #1
    \int_compare:nF { `a <= `#1 <= `z }
      {
        \int_compare:nF { `A <= `#1 <= `Z }
          { \prg_break:n \prg_return_true: }
      }
    \__fp_id_if_invalid_aux:N
  }
```

(*End of definition for* `\__fp_id_if_invalid:nTF` *and* `\__fp_id_if_invalid_aux:N`.)

81.8 Declaring variables and assigning values

`\__fp_variable_o:w`

We do not use `\exp_last_unbraced:Nv` to extract the value of `\l__fp_variable_#1_fp` because in `\fp_set_variable:nn` we define this `fp` variable to be something which `f`-expands to an actual floating point, rather than a genuine floating point.

```
\cs_new:Npn \__fp_variable_o:w #1 @ #2
  {
    \fp_if_exist:cTF { l__fp_variable_#1_fp }
      {
        \exp_last_unbraced:Nf \__fp_exp_after_array_f:w
          { \use:c { l__fp_variable_#1_fp } } \s__fp_expr_stop
        \exp_after:wN \exp_stop_f: #2
      }
      {
        \token_if_eq_meaning:NNTF #2 \prg_do_nothing:
          {
            \s__fp_symbolic \__fp_symbolic_chk:w
              \__fp_variable_o:w #1 , { } ;
          }
          {
            \exp_after:wN \s__fp_symbolic
            \exp_after:wN \__fp_symbolic_chk:w
            \exp_after:wN \__fp_variable_o:w
            \exp:w
            \__fp_exp_after_symbolic_loop:N #1
              { , \exp:w \use_none:nn }
            \exp_after:wN \exp_end:
            \exp_after:wN { \exp_after:wN } \exp_after:wN ;
            #2
          }
      }
  }
```

(*End of definition for* `\__fp_variable_o:w`.)

`\__fp_variable_set_parsing:Nn`
`\__fp_variable_set_parsing_aux:NNn`

```
\cs_new_protected:Npn \__fp_variable_set_parsing:Nn #1#2
  {
    \cs_set:Npn \__fp_tmp:w
      {
        \__fp_exp_after_symbolic_f:nw { \__fp_parse_infix:NN }
        \s__fp_symbolic \__fp_symbolic_chk:w
          \__fp_variable_o:w #2 , { } ;
      }
    \exp_args:NNc \__fp_variable_set_parsing_aux:NNn #1
      { __fp_parse_word_#2:N } {#2}
  }
\cs_new_protected:Npn \__fp_variable_set_parsing_aux:NNn #1#2#3
  {
    \cs_if_eq:NNF #2 \__fp_tmp:w
      {
        \cs_if_exist:NTF #2
          {
            \msg_warning:nnnn
              { fp } { id-used-elsewhere } {#3} { variable }
            #1 #2 \__fp_tmp:w
          }
          {
            \cs_new_eq:NN #2 \scan_stop: % to declare the function
            #1 #2 \__fp_tmp:w
          }
      }
  }
```

(*End of definition for* `\__fp_variable_set_parsing:Nn` *and* `\__fp_variable_set_parsing_aux:NNn`.)

`\fp_clear_variable:n`
`\__fp_clear_variable:n`
`\__fp_clear_variable_aux:n`

We need local undefining, so have to do it low-level. `\__fp_clear_variable_aux:n` is needed by `\__fp_set_function:Nnnn` to skip `\__fp_id_if_invalid:nTF`.

```
\cs_new_protected:Npn \fp_clear_variable:n #1
  {
    \exp_args:No \__fp_clear_variable:n { \tl_to_str:n {#1} }
  }
\cs_new_protected:Npn \__fp_clear_variable:n #1
  {
    \__fp_id_if_invalid:nTF {#1}
      { \msg_error:nnn { fp } { id-invalid } {#1} }
      { \__fp_clear_variable_aux:n {#1} }
  }
\cs_new_protected:Npn \__fp_clear_variable_aux:n #1
  {
    \cs_set_eq:cN { l__fp_variable_#1_fp } \tex_undefined:D
    \__fp_variable_set_parsing:Nn \cs_set_eq:NN {#1}
  }
```

(*End of definition for* `\fp_clear_variable:n`*,* `\__fp_clear_variable:n`*, and* `\__fp_clear_variable_aux:n`*. This function is documented on page 267.*)

`\fp_new_variable:n`
`\__fp_new_variable:n`

Check that `#1` is a valid identifier. If the identifier is already in use, complain. Then set `\__fp_parse_word_#1:N` to use `\__fp_variable_o:w`.

```
\cs_new_protected:Npn \fp_new_variable:n #1
```

```
  {
    \exp_args:No \__fp_new_variable:n { \tl_to_str:n {#1} }
  }
\cs_new_protected:Npn \__fp_new_variable:n #1
  {
    \__fp_id_if_invalid:nTF {#1}
      { \msg_error:nnn { fp } { id-invalid } {#1} }
      {
        \cs_if_exist:cT { __fp_parse_word_#1:N }
        {
          \msg_error:nnn
            { fp } { id-already-defined } {#1}
          \cs_undefine:c { __fp_parse_word_#1:N }
          \cs_set_eq:cN { l__fp_variable_#1_fp } \tex_undefined:D
        }
      \__fp_variable_set_parsing:Nn \cs_gset_eq:NN {#1}
      }
  }
```

(*End of definition for* `\fp_new_variable:n` *and* `\__fp_new_variable:n`*. This function is documented on page 266.*)

`\l__fp_symbolic_flag`
`\fp_set_variable:nn`
`\__fp_set_variable:nn`

Refuse invalid identifiers. If the variable does not exist yet, define it just as in `\fp_new_variable:n` (but without unnecessary checks). Then evaluate `#2`. If the result contains the identifier `#1`, we would later get a loop in cases such as

```
\fp_set_variable:nn {A} {A}
\fp_show:n {A}
```

To detect this, define `\l__fp_variable_#1_fp` to raise an internal flag and evaluate to `nan`. Then re-evaluate `\l__fp_symbolic_fp`, and store the result in `#1`. If the flag is raised, `#1` was present in `\l__fp_symbolic_fp`. In all cases, the `#1`-free result ends up in `\l__fp_variable_#1_fp`.

```
\flag_new:N \l__fp_symbolic_flag
\cs_new_protected:Npn \fp_set_variable:nn #1
  {
    \exp_args:No \__fp_set_variable:nn { \tl_to_str:n {#1} }
  }
\cs_new_protected:Npn \__fp_set_variable:nn #1#2
  {
    \__fp_id_if_invalid:nTF {#1}
      { \msg_error:nnn { fp } { id-invalid } {#1} }
      {
        \__fp_variable_set_parsing:Nn \cs_set_eq:NN {#1}
        \fp_set:Nn \l__fp_symbolic_fp {#2}
        \cs_set_nopar:cpn { l__fp_variable_#1_fp }
          { \flag_ensure_raised:N \l__fp_symbolic_flag \c_nan_fp }
        \flag_clear:N \l__fp_symbolic_flag
        \fp_set:cn { l__fp_variable_#1_fp } { \l__fp_symbolic_fp }
        \flag_if_raised:NT \l__fp_symbolic_flag
          {
            \msg_error:nneee { fp } { id-loop }
              { #1 }
              { \tl_to_str:n {#2} }
```

```
              { \fp_to_tl:N \l__fp_symbolic_fp }
          }
      }
  }
```

(*End of definition for* `\l__fp_symbolic_flag`*,* `\fp_set_variable:nn`*, and* `\__fp_set_variable:nn`*. This variable is documented on page 266.*)

81.9 Messages

```
\msg_new:nnnn { fp } { id-invalid }
  { Floating~point~identifier~'#1'~invalid. }
  {
    LaTeX~has~been~asked~to~create~a~new~floating~point~identifier~'#1'~
    but~this~may~only~contain~ASCII~letters.
  }
\msg_new:nnnn { fp } { id-already-defined }
  { Floating~point~identifier~'#1'~already~defined. }
  {
    LaTeX~has~been~asked~to~create~a~new~floating~point~identifier~'#1'~
    but~this~name~has~already~been~used~elsewhere.
  }
\msg_new:nnnn { fp } { id-used-elsewhere }
  { Floating~point~identifier~'#1'~already~used~for~something~else. }
  {
    LaTeX~has~been~asked~to~create~a~new~floating~point~identifier~'#1'~
    but~this~name~is~used,~and~is~not~a~user-defined~#2.
  }
\msg_new:nnnn { fp } { id-loop }
  { Variable~'#1'~used~in~the~definition~of~'#1'. }
  {
    LaTeX~has~been~asked~to~set~the~floating~point~identifier~'#1'~
    to~the~expression~'#2'.~Evaluating~this~expression~yields~'#3',~
    which~contains~'#1'~itself.
  }
```

81.10 Road-map

The following functions are not implemented: `min`, `max`, `?:`, comparisons, `round`, `atan`, `acot`.

⟨/package⟩

Chapter 82

l3fp-functions implementation

```
⟨*package⟩
⟨@@=fp⟩
```

82.1 Declaring functions

\fp_new_function:n
__fp_new_function:n

```
\cs_new_protected:Npn \fp_new_function:n #1
  { \exp_args:No \__fp_new_function:n { \tl_to_str:n {#1} } }
\cs_new_protected:Npn \__fp_new_function:n #1
  {
    \__fp_id_if_invalid:nTF {#1}
      { \msg_error:nnn { fp } { id-invalid } {#1} }
      {
        \cs_if_exist:cT { __fp_parse_word_#1:N }
          {
            \msg_error:nnn
              { fp } { id-already-defined } {#1}
            \cs_undefine:c { __fp_parse_word_#1:N }
            \cs_undefine:c { __fp_#1_o:w }
          }
        \__fp_function_set_parsing:Nn \cs_gset_eq:NN {#1}
      }
  }
```

(*End of definition for* `\fp_new_function:n` *and* `\__fp_new_function:n`. *This function is documented on page* *267*.)

__fp_function_set_parsing:Nn
__fp_function_set_parsing_aux:NNn

```
\cs_new_protected:Npn \__fp_function_set_parsing:Nn #1#2
  {
    \exp_args:NNc \__fp_function_set_parsing_aux:NNn #1
      { __fp_parse_word_#2:N } {#2}
  }
\cs_new_protected:Npn \__fp_function_set_parsing_aux:NNn #1#2#3
  {
    \cs_set:Npe \__fp_tmp:w
      {
        \exp_not:N \__fp_parse_function:NNN
```

```
        \exp_not:N \__fp_function_o:w
        \exp_not:c { __fp_#3_o:w }
      }
    \cs_if_eq:NNF #2 \__fp_tmp:w
      {
        \cs_if_exist:NTF #2
          {
            \msg_warning:nnnn
              { fp } { id-used-elsewhere } {#3} { function }
            #1 #2 \__fp_tmp:w
          }
          {
            \cs_new_eq:NN #2 \scan_stop: % to declare the function
            #1 #2 \__fp_tmp:w
          }
      }
  }
```

(*End of definition for* `\__fp_function_set_parsing:Nn` *and* `\__fp_function_set_parsing_aux:NNn`.)

`\__fp_function_o:w`

```
\cs_new:Npn \__fp_function_o:w #1#2 @
  {
    \cs_if_exist:NTF #1
      { #1 #2 @ }
      {
        \exp_after:wN \s__fp_symbolic
        \exp_after:wN \__fp_symbolic_chk:w
        \exp_after:wN \__fp_function_o:w
        \exp_after:wN #1
        \exp_after:wN ,
        \exp_after:wN {
          \exp:w \exp_end_continue_f:w
          \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
          \exp_after:wN
        }
        \exp_after:wN ;
      }
  }
```

(*End of definition for* `\__fp_function_o:w`.)

82.2 Defining functions by their expression

`\l__fp_function_arg_int`

Labels the arguments of a function being defined.

```
\int_new:N \l__fp_function_arg_int
```

(*End of definition for* `\l__fp_function_arg_int`.)

`\fp_set_function:nnn`
`\__fp_set_function:Nnnn`

`\fp_set_function:nnn {⟨identifier⟩} {⟨comma-list of variables⟩} {⟨expression⟩}`

Defines the ⟨*identifier*⟩ to stand for a function which expects some arguments defined by the ⟨*comma-list of variables*⟩, and evaluates to the ⟨*expression*⟩.

```
\cs_new_protected:Npn \fp_set_function:nnn #1
  {
    \exp_args:NNo \__fp_set_function:Nnnn \cs_set_eq:cN
      { \tl_to_str:n {#1} }
  }
\cs_new_protected:Npn \__fp_set_function:Nnnn #1#2#3#4
  {
    \__fp_id_if_invalid:nTF {#2}
      { \msg_error:nnn { fp } { id-invalid } {#2} }
      {
        \cs_if_exist:cF { __fp_parse_word_#2:N }
          { \__fp_function_set_parsing:Nn \cs_set_eq:NN {#2} }
        \group_begin:
          \int_zero:N \l__fp_function_arg_int
          \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#3} }
            {
              \int_incr:N \l__fp_function_arg_int
              \exp_args:Ne \__fp_clear_variable_aux:n
                {
                  \c_underscore_str \tex_romannumeral:D \l__fp_function_arg_int
                }
              \fp_clear_variable:n {##1}
              \cs_set_nopar:cpe { l__fp_variable_##1_fp }
                {
                  \exp_not:N \s__fp_symbolic
                  \exp_not:N \__fp_symbolic_chk:w
                  \exp_not:N \__fp_function_arg_o:w
                  \int_use:N \l__fp_function_arg_int
                  ########1 , { } ;
                }
            }
          \cs_set:Npn \__fp_function_arg_o:w ##1 @
            {
              \exp_after:wN \s__fp_symbolic
              \exp_after:wN \__fp_symbolic_chk:w
              \exp_after:wN \__fp_function_arg_o:w
              \tex_romannumeral:D
              \__fp_exp_after_symbolic_loop:N ##1
                { , \tex_romannumeral:D \use_none:nn }
              \exp_after:wN \c_zero_int
              \exp_after:wN { \exp_after:wN } \exp_after:wN ;
            }
          \fp_set:Nn \l__fp_symbolic_fp {#4}
          \use:e
            {
              \exp_not:n { \cs_gset:Npn \__fp_tmp:w ##1 }
                { \exp_not:o { \l__fp_symbolic_fp } }
            }
          \use:e
            {
              \exp_not:n { \cs_gset:Npn \__fp_tmp:w ##1 @ }
                {
                  \exp_not:N \__fp_exp_after_symbolic_f:nw
                  \exp_not:n { { \exp_after:wN \exp_stop_f: } }
```

```
                    \exp_not:o { \__fp_tmp:w { . , {##1} } }
                  }
              }
          \group_end:
          #1 { __fp_#2_o:w } \__fp_tmp:w
        }
    }
```

__fp_function_arg_o:w
__fp_function_arg_few:w
__fp_function_arg_get:w

```
\cs_new:Npn \__fp_function_arg_o:w #1. #2
  {
    \if_meaning:w @ #2
      \exp_after:wN \__fp_function_arg_few:w
    \fi:
    \if_int_compare:w #1 = \c_one_int
      \exp_after:wN \__fp_function_arg_get:w
    \fi:
    \__fp_use_i_until_s:nw
      {
        \exp_after:wN \__fp_function_arg_o:w
        \int_value:w \int_eval:n { #1 - 1 } .
      }
      #2
  }
\cs_new:Npn \__fp_function_arg_few:w #1 @ { \exp_after:wN \c_nan_fp }
\cs_new:Npn \__fp_function_arg_get:w #1#2#3; #4 @
  {
    \__fp_exp_after_array_f:w #3; \s__fp_expr_stop
    \exp_after:wN \exp_stop_f:
  }
```

(*End of definition for* `\fp_set_function:nnn` *and others. This function is documented on page 267.*)

\fp_clear_function:n
__fp_clear_function:n

```
\cs_new_protected:Npn \fp_clear_function:n #1
  { \exp_args:No \__fp_clear_function:n { \tl_to_str:n {#1} } }
\cs_new_protected:Npn \__fp_clear_function:n #1
  {
    \__fp_id_if_invalid:nTF {#1}
      { \msg_error:nnn { fp } { id-invalid } {#1} }
      {
        \cs_set_eq:cN { __fp_#1_o:w } \tex_undefine:D
        \__fp_function_set_parsing:Nn \cs_set_eq:NN {#1}
      }
  }
```

(*End of definition for* `\fp_clear_function:n` *and* `\__fp_clear_function:n`*. This function is documented on page 267.*)

```
⟨/package⟩
```

Chapter 83

l3fparray implementation

```
⟨*package⟩
⟨@@=fp⟩
```

In analogy to l3intarray it would make sense to have `<@@=fparray>`, but we need direct access to `\__fp_parse:n` from l3fp-parse, and a few other (less crucial) internals of the l3fp family.

83.1 Allocating arrays

There are somewhat more than $(2^{31}-1)^2$ floating point numbers so we store each floating point number as three entries in integer arrays. To avoid having to multiply indices by three or to add 1 etc, a floating point array is just a token list consisting of three tokens: integer arrays of the same size.

`\g__fp_array_int` Used to generate unique names for the three integer arrays.

```
\int_new:N \g__fp_array_int
```

(*End of definition for* `\g__fp_array_int`.)

`\l__fp_array_loop_int` Used to loop in `\__fp_array_gzero:N`.

```
\int_new:N \l__fp_array_loop_int
```

(*End of definition for* `\l__fp_array_loop_int`.)

`\fparray_new:Nn`
`\fparray_new:cn`
`\__fp_array_new:nNNN`

Build a three-token token list, then define all three tokens to be integer arrays of the same size. No need to initialize the data: the integer arrays start with zeros, and three zeros denote precisely `\c_zero_fp`, as we want.

```
\cs_new_protected:Npn \fparray_new:Nn #1#2
  {
    \tl_new:N #1
    \prg_replicate:nn { 3 }
      {
        \int_gincr:N \g__fp_array_int
        \exp_args:NNc \tl_gput_right:Nn #1
          { g__fp_array_ \__fp_int_to_roman:w \g__fp_array_int _intarray }
      }
    \exp_last_unbraced:Nfo \__fp_array_new:nNNNN
      { \int_eval:n {#2} } #1 #1
```

```
  }
\cs_generate_variant:Nn \fparray_new:Nn { c }
\cs_new_protected:Npn \__fp_array_new:nNNNN #1#2#3#4#5
  {
    \int_compare:nNnTF {#1} < 0
      {
        \msg_error:nnn { kernel } { negative-array-size } {#1}
        \cs_undefine:N #1
        \int_gsub:Nn \g__fp_array_int { 3 }
      }
      {
        \intarray_new:Nn #2 {#1}
        \intarray_new:Nn #3 {#1}
        \intarray_new:Nn #4 {#1}
      }
  }
```

(*End of definition for* `\fparray_new:Nn` *and* `\__fp_array_new:nNNN`. *This function is documented on page 282.*)

`\fparray_count:N`
`\fparray_count:c`

Size of any of the intarrays, here we pick the third.

```
\cs_new:Npn \fparray_count:N #1
  {
    \exp_after:wN \use_i:nnn
    \exp_after:wN \intarray_count:N #1
  }
\cs_generate_variant:Nn \fparray_count:N { c }
```

(*End of definition for* `\fparray_count:N`. *This function is documented on page 283.*)

83.2 Array items

`\__fp_array_bounds:NNnTF`
`\__fp_array_bounds_error:NNn`

See the l3intarray analogue: only names change. The functions `\fparray_gset:Nnn` and `\fparray_item:Nn` share bounds checking. The `T` branch is used if `#3` is within bounds of the array `#2`.

```
\cs_new:Npn \__fp_array_bounds:NNnTF #1#2#3#4#5
  {
    \if_int_compare:w 1 > #3 \exp_stop_f:
      \__fp_array_bounds_error:NNn #1 #2 {#3}
      #5
    \else:
      \if_int_compare:w #3 > \fparray_count:N #2 \exp_stop_f:
        \__fp_array_bounds_error:NNn #1 #2 {#3}
        #5
      \else:
        #4
      \fi:
    \fi:
  }
\cs_new:Npn \__fp_array_bounds_error:NNn #1#2#3
  {
    #1 { kernel } { out-of-bounds }
      { \token_to_str:N #2 } {#3} { \fparray_count:N #2 }
  }
```

(*End of definition for* `\__fp_array_bounds:NNnTF` *and* `\__fp_array_bounds_error:NNn`.)

`\fparray_gset:Nnn`
`\fparray_gset:cnn`
`\__fp_array_gset:NNNNww`
`\__fp_array_gset:w`
`\__fp_array_gset_recover:Nw`
`\__fp_array_gset_special:nnNNN`
`\__fp_array_gset_normal:w`

Evaluate, then store exponent in one intarray, sign and 8 digits of mantissa in the next, and 8 trailing digits in the last.

```
\cs_new_protected:Npn \fparray_gset:Nnn #1#2#3
  {
    \exp_after:wN \exp_after:wN
    \exp_after:wN \__fp_array_gset:NNNNww
    \exp_after:wN #1
    \exp_after:wN #1
    \int_value:w \int_eval:n {#2} \exp_after:wN ;
    \exp:w \exp_end_continue_f:w \__fp_parse:n {#3}
  }
\cs_generate_variant:Nn \fparray_gset:Nnn { c }
\cs_new_protected:Npn \__fp_array_gset:NNNNww #1#2#3#4#5 ; #6 ;
  {
    \__fp_array_bounds:NNnTF \msg_error:nneee #4 {#5}
      {
        \exp_after:wN \__fp_change_func_type:NNN
          \__fp_use_i_until_s:nw #6 ;
          \__fp_array_gset:w
          \__fp_array_gset_recover:Nw
        #6 ; {#5} #1 #2 #3
      }
      { }
  }
\cs_new_protected:Npn \__fp_array_gset_recover:Nw #1#2 ;
  {
    \__fp_error:nffn { unknown-type } { \tl_to_str:n { #2 ; } } { } { }
    \exp_after:wN #1 \c_nan_fp
  }
\cs_new_protected:Npn \__fp_array_gset:w \s__fp \__fp_chk:w #1#2
  {
    \if_case:w #1 \exp_stop_f:
         \__fp_case_return:nw { \__fp_array_gset_special:nnNNN {#2} }
    \or: \exp_after:wN \__fp_array_gset_normal:w
    \or: \__fp_case_return:nw { \__fp_array_gset_special:nnNNN { #2 3 } }
    \or: \__fp_case_return:nw { \__fp_array_gset_special:nnNNN { 1 } }
    \fi:
    \s__fp \__fp_chk:w #1 #2
  }
\cs_new_protected:Npn \__fp_array_gset_normal:w
  \s__fp \__fp_chk:w 1 #1 #2 #3#4#5 ; #6#7#8#9
  {
    \__kernel_intarray_gset:Nnn #7 {#6} {#2}
    \__kernel_intarray_gset:Nnn #8 {#6}
      { \if_meaning:w 2 #1 3 \else: 1 \fi: #3#4 }
    \__kernel_intarray_gset:Nnn #9 {#6} { 1 \use:nn #5 }
  }
\cs_new_protected:Npn \__fp_array_gset_special:nnNNN #1#2#3#4#5
  {
    \__kernel_intarray_gset:Nnn #3 {#2} {#1}
    \__kernel_intarray_gset:Nnn #4 {#2} {0}
    \__kernel_intarray_gset:Nnn #5 {#2} {0}
```

```
  }
```

(*End of definition for* `\fparray_gset:Nnn` *and others. This function is documented on page* 282.)

`\fparray_gzero:N`
`\fparray_gzero:c`

```
\cs_new_protected:Npn \fparray_gzero:N #1
  {
    \int_zero:N \l__fp_array_loop_int
    \prg_replicate:nn { \fparray_count:N #1 }
      {
        \int_incr:N \l__fp_array_loop_int
        \exp_after:wN \__fp_array_gset_special:nnNNN
        \exp_after:wN 0
        \exp_after:wN \l__fp_array_loop_int
        #1
      }
  }
\cs_generate_variant:Nn \fparray_gzero:N { c }
```

(*End of definition for* `\fparray_gzero:N`. *This function is documented on page* 282.)

`\fparray_item:Nn`
`\fparray_item:cn`
`\fparray_item_to_tl:Nn`
`\fparray_item_to_tl:cn`
`\__fp_array_item:NwN`
`\__fp_array_item:NNNnN`
`\__fp_array_item:N`
`\__fp_array_item:w`
`\__fp_array_item_special:w`
`\__fp_array_item_normal:w`

```
\cs_new:Npn \fparray_item:Nn #1#2
  {
    \exp_after:wN \__fp_array_item:NwN
    \exp_after:wN #1
    \int_value:w \int_eval:n {#2} ;
    \__fp_to_decimal:w
  }
\cs_generate_variant:Nn \fparray_item:Nn { c }
\cs_new:Npn \fparray_item_to_tl:Nn #1#2
  {
    \exp_after:wN \__fp_array_item:NwN
    \exp_after:wN #1
    \int_value:w \int_eval:n {#2} ;
    \__fp_to_tl:w
  }
\cs_generate_variant:Nn \fparray_item_to_tl:Nn { c }
\cs_new:Npn \__fp_array_item:NwN #1#2 ; #3
  {
    \__fp_array_bounds:NNnTF \msg_expandable_error:nnfff #1 {#2}
      { \exp_after:wN \__fp_array_item:NNNnN #1 {#2} #3 }
      { \exp_after:wN #3 \c_nan_fp }
  }
\cs_new:Npn \__fp_array_item:NNNnN #1#2#3#4
  {
    \exp_after:wN \__fp_array_item:N
    \int_value:w \__kernel_intarray_item:Nn #2 {#4} \exp_after:wN ;
    \int_value:w \__kernel_intarray_item:Nn #3 {#4} \exp_after:wN ;
    \int_value:w \__kernel_intarray_item:Nn #1 {#4} ;
  }
\cs_new:Npn \__fp_array_item:N #1
  {
    \if_meaning:w 0 #1 \exp_after:wN \__fp_array_item_special:w \fi:
    \__fp_array_item:w #1
```

```
  }
\cs_new:Npn \__fp_array_item:w #1 #2#3#4#5 #6 ; 1 #7 ;
  {
    \exp_after:wN \__fp_array_item_normal:w
    \int_value:w \if_meaning:w #1 1 0 \else: 2 \fi: \exp_stop_f:
    #7 ; {#2#3#4#5} {#6} ;
  }
\cs_new:Npn \__fp_array_item_special:w #1 ; #2 ; #3 ; #4
  {
    \exp_after:wN #4
    \exp:w \exp_end_continue_f:w
    \if_case:w #3 \exp_stop_f:
         \exp_after:wN \c_zero_fp
    \or: \exp_after:wN \c_nan_fp
    \or: \exp_after:wN \c_minus_zero_fp
    \or: \exp_after:wN \c_inf_fp
    \else: \exp_after:wN \c_minus_inf_fp
    \fi:
  }
\cs_new:Npn \__fp_array_item_normal:w #1 #2#3#4#5 #6 ; #7 ; #8 ; #9
  { #9 \s__fp \__fp_chk:w 1 #1 {#8} #7 {#2#3#4#5} {#6} ; }
```

(*End of definition for* `\fparray_item:Nn` *and others. These functions are documented on page 283.*)

`\fparray_if_exist_p:N`
`\fparray_if_exist_p:c`
`\fparray_if_exist:NTF`
`\fparray_if_exist:cTF`

Copies of the cs functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \fparray_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \fparray_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End of definition for* `\fparray_if_exist:NTF`. *This function is documented on page 283.*)

```
⟨/package⟩
```

Chapter 84

l3bitset implementation

```
⟨*package⟩
⟨@@=bitset⟩
```

Transitional support.

```
\cs_if_exist:NT \@expl@finalise@setup@@
  {
    \tl_gput_right:Nn \@expl@finalise@setup@@
      { \declare@file@substitution { l3bitset.sty } { null.tex } }
  }
```

A bitset is a string variable.

\bitset_new:N
\bitset_new:c
\bitset_new:Nn
\bitset_new:cn

```
\cs_new_protected:Npn \bitset_new:N #1
  {
    \__kernel_chk_if_free_cs:N #1
    \cs_gset_eq:NN #1 \c_zero_str
    \prop_new:c { g__bitset_ \cs_to_str:N #1 _name_prop }
  }
\cs_new_protected:Npn \bitset_new:Nn #1 #2
  {
    \__kernel_chk_if_free_cs:N #1
    \cs_gset_eq:NN #1 \c_zero_str
    \prop_new:c { g__bitset_ \cs_to_str:N #1 _name_prop }
    \prop_gset_from_keyval:cn
      { g__bitset_ \cs_to_str:N #1 _name_prop }
      {#2}
  }
\cs_generate_variant:Nn \bitset_new:N { c }
\cs_generate_variant:Nn \bitset_new:Nn { c }
```

(*End of definition for* `\bitset_new:N` *and* `\bitset_new:Nn`*. These functions are documented on page* *285*.)

\bitset_addto_named_index:Nn

```
\cs_new_protected:Npn \bitset_addto_named_index:Nn #1#2
  {
    \prop_gput_from_keyval:cn
      { g__bitset_ \cs_to_str:N #1 _name_prop } { #2 }
  }
```

(*End of definition for* `\bitset_addto_named_index:Nn`. *This function is documented on page 285.*)

`\bitset_if_exist_p:N`
`\bitset_if_exist_p:c`
`\bitset_if_exist:NTF`
`\bitset_if_exist:cTF`

Existence tests.

```
\prg_new_eq_conditional:NNn
  \bitset_if_exist:N \str_if_exist:N { p , T , F , TF }
\prg_new_eq_conditional:NNn
  \bitset_if_exist:c \str_if_exist:c { p , T , F , TF }
```

(*End of definition for* `\bitset_if_exist:NTF`. *This function is documented on page 286.*)

`\__bitset_set_true:Nn`
`\__bitset_gset_true:Nn`
`\__bitset_set_false:Nn`
`\__bitset_gset_false:Nn`
`\__bitset_set:NNnN`

The internal command uses only numbers (integer expressions) for the position. A bit is set by either extending the string or by splitting it and then inserting an 1. It is not checked if the value was already 1.

```
\cs_new_protected:Npn \__bitset_set_true:Nn #1#2
  { \__bitset_set:NNnN \str_set:Ne #1 {#2} 1 }
\cs_new_protected:Npn \__bitset_gset_true:Nn #1#2
  { \__bitset_set:NNnN \str_gset:Ne #1 {#2} 1 }
\cs_new_protected:Npn \__bitset_set_false:Nn #1#2
  { \__bitset_set:NNnN \str_set:Ne #1 {#2} 0 }
\cs_new_protected:Npn \__bitset_gset_false:Nn #1#2
  { \__bitset_set:NNnN \str_gset:Ne #1 {#2} 0 }
\cs_new_protected:Npn \__bitset_set:NNnN #1#2#3#4
  {
    \int_compare:nNnT {#3} > { 0 }
      {
        \int_compare:nNnTF { \str_count:N #2 } < {#3}
          {
            #1 #2
              {
                #4
                \prg_replicate:nn { #3 - \str_count:N #2 - 1 } { 0 }
                #2
              }
          }
          {
            #1 #2
              {
                \str_range:Nnn #2 { 1 } { -1 - (#3) }
                #4
                \str_range:Nnn #2 { 1 - (#3) } { -1 }
              }
          }
      }
  }
```

(*End of definition for* `\__bitset_set_true:Nn` *and others.*)

`\l__bitset_internal_int`

```
\int_new:N \l__bitset_internal_int
```

(*End of definition for* `\l__bitset_internal_int`.)

__bitset_test_digits:nTF
__bitset_test_digits_end:n
__bitset_test_digits:w

https://chat.stackexchange.com/transcript/message/56878159#56878159

```
\prg_new_protected_conditional:Npnn \__bitset_test_digits:n #1 { TF }
  {
    \tex_afterassignment:D \__bitset_test_digits:w
    \l__bitset_internal_int = 0 \tl_trim_spaces_apply:nN {#1} \tl_to_str:n
    \__bitset_test_digits_end:
    \use_i:nnn \if_false:
    \__bitset_test_digits_end:
    \if_int_compare:w \c_zero_int < \l__bitset_internal_int
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new_eq:NN \__bitset_test_digits_end: \exp_stop_f:
\cs_new_protected:Npn \__bitset_test_digits:w #1 \__bitset_test_digits_end: { }
```

(*End of definition for* `\__bitset_test_digits:nTF`, `\__bitset_test_digits_end:n`, *and* `\__bitset_test_digits:w`.)

\bitset_set_true:Nn
\bitset_set_true:cn
\bitset_gset_true:Nn
\bitset_gset_true:cn
\bitset_set_false:Nn
\bitset_set_false:cn
\bitset_gset_false:Nn
\bitset_gset_false:cn
__bitset_set_aux:NNn

The user commands must first translate the argument to an index number.

```
\cs_new_protected:Npn \bitset_set_true:Nn #1#2
  { \__bitset_set:NNn \__bitset_set_true:Nn #1 {#2} }
\cs_new_protected:Npn \bitset_gset_true:Nn #1#2
  { \__bitset_set:NNn \__bitset_gset_true:Nn #1 {#2} }
\cs_new_protected:Npn \bitset_set_false:Nn #1#2
  { \__bitset_set:NNn \__bitset_set_false:Nn #1 {#2} }
\cs_new_protected:Npn \bitset_gset_false:Nn #1#2
  { \__bitset_set:NNn \__bitset_gset_false:Nn #1 {#2} }
\cs_new_protected:Npn \__bitset_set:NNn #1#2#3
  {
    \prop_if_in:cnTF { g__bitset_ \cs_to_str:N #2 _name_prop } {#3}
      {
        #1 #2
          {
            \prop_item:cn  { g__bitset_ \cs_to_str:N #2 _name_prop } {#3}
          }
      }
      {
        \__bitset_test_digits:nTF {#3}
          {
            #1 #2 {#3}
            \prop_gput:cnn { g__bitset_ \cs_to_str:N #2 _name_prop } {#3} {#3}
          }
          {
            \msg_warning:nnee { bitset } { unknown-name }
              { \token_to_str:N #2 }
              { \tl_to_str:n {#3} }
          }
      }
  }
\cs_generate_variant:Nn \bitset_set_true:Nn  { c }
\cs_generate_variant:Nn \bitset_gset_true:Nn { c }
\cs_generate_variant:Nn \bitset_set_false:Nn  { c }
\cs_generate_variant:Nn \bitset_gset_false:Nn { c }
```

(*End of definition for* `\bitset_set_true:Nn` *and others. These functions are documented on page 286.*)

`\bitset_clear:N`
`\bitset_clear:c`
`\bitset_gclear:N`
`\bitset_gclear:c`

```
\cs_new_protected:Npn \bitset_clear:N #1
  {
    \str_set_eq:NN #1 \c_zero_str
  }
\cs_new_protected:Npn \bitset_gclear:N #1
  {
    \str_gset_eq:NN #1 \c_zero_str
  }
\cs_generate_variant:Nn \bitset_clear:N { c }
\cs_generate_variant:Nn \bitset_gclear:N { c }
```

(*End of definition for* `\bitset_clear:N` *and* `\bitset_gclear:N`*. These functions are documented on page 286.*)

`\bitset_to_arabic:N`
`\bitset_to_arabic:c`
`\bitset_to_bin:N`
`\bitset_to_bin:c`
`\__bitset_to_int:nN`

The naming of the commands follow the names in the `int` module. `\bitset_to_arabic:N` uses `\int_from_bin:n` if the string is shorter than 32 and the slower `\fp_eval:n` for larger bitsets.

```
\cs_new:Npn \bitset_to_arabic:N #1
  {
    \int_compare:nNnTF { \str_count:N #1 } < { 32 }
      { \exp_args:No \int_from_bin:n {#1} }
      {
        \exp_after:wN \__bitset_to_int:nN \exp_after:wN 0
        #1 \q_recursion_tail \q_recursion_stop
      }
  }
\cs_new:Npn \__bitset_to_int:nN #1#2
  {
    \quark_if_recursion_tail_stop_do:Nn #2 {#1}
    \exp_args:Nf \__bitset_to_int:nN { \fp_eval:n { #1 * 2 + #2 } }
  }
\cs_new:Npn \bitset_to_bin:N #1
  {
    #1
  }
\cs_generate_variant:Nn \bitset_to_arabic:N  { c }
\cs_generate_variant:Nn \bitset_to_bin:N { c }
```

(*End of definition for* `\bitset_to_arabic:N`*,* `\bitset_to_bin:N`*, and* `\__bitset_to_int:nN`*. These functions are documented on page 287.*)

`\bitset_item:Nn`
`\bitset_item:cn`

All bits that have been set at anytime have an entry in the prop, so we can take everything else as 0.

```
\cs_new:Npn \bitset_item:Nn #1#2
  {
    \prop_if_in:cnTF { g__bitset_ \cs_to_str:N #1 _name_prop } {#2}
      {
        \int_eval:n
          {
            \str_item:Nn #1
              { 0 - ( \prop_item:cn { g__bitset_ \cs_to_str:N #1 _name_prop } {#2} ) }
```

```
              +0
            }
        }
        {
          0
        }
    }
\cs_generate_variant:Nn \bitset_item:Nn { c }
```

(*End of definition for* `\bitset_item:Nn`. *This function is documented on page 286.*)

`\bitset_show:N`
`\bitset_show:c`
`\bitset_log:N`
`\bitset_log:c`

```
\cs_new_protected:Npn   \bitset_show:N { \__bitset_show:NN \msg_show:nneeee }
\cs_generate_variant:Nn \bitset_show:N { c }
\cs_new_protected:Npn   \bitset_log:N  { \__bitset_show:NN \msg_log:nneeee }
\cs_generate_variant:Nn \bitset_log:N  { c }
\cs_new_protected:Npn \__bitset_show:NN #1#2
  {
    \__kernel_chk_defined:NT #2
      {
        #1 { bitset } { show }
          { \token_to_str:N #2 }
          { \bitset_to_bin:N #2 }
          { \bitset_to_arabic:N #2 }
          { }
      }
  }
```

(*End of definition for* `\bitset_show:N` *and* `\bitset_log:N`. *These functions are documented on page 287.*)

`\bitset_show_named_index:N`
`\bitset_show_named_index:c`
`\bitset_log_named_index:N`
`\bitset_log_named_index:c`

```
\cs_new_protected:Npn \bitset_show_named_index:N
  { \__bitset_show_named_index:NN \msg_show:nneeee }
\cs_generate_variant:Nn \bitset_show_named_index:N { c }
\cs_new_protected:Npn \bitset_log_named_index:N
  { \__bitset_show_named_index:NN \msg_log:nneeee }
\cs_generate_variant:Nn \bitset_log_named_index:N { c }
\cs_new_protected:Npn \__bitset_show_named_index:NN #1#2
  {
    \__kernel_chk_defined:NT #2
      {
        #1 { bitset } { show-names }
          { \token_to_str:N #2 }
          { \prop_map_function:cN { g__bitset_ \cs_to_str:N #2 _name_prop } \msg_show_item:
          { } { }
      }
  }
```

(*End of definition for* `\bitset_show_named_index:N` *and* `\bitset_log_named_index:N`. *These functions are documented on page 287.*)

84.1 Messages

```
\msg_new:nnn { bitset } { show }
  {
    The~bitset~#1~has~the~representation: \\
    >~binary:~#2  \\
    >~arabic:~#3 .
  }
\msg_new:nnn { bitset } { show-names }
  {
    The~bitset~#1~
    \tl_if_empty:nTF {#2}
      { knows~no~names~yet \\>~ . }
      { knows~the~name/index~pairs~(without~outer~braces): #2 . }
  }
\msg_new:nnn { bitset } { unknown-name }
  { The~name~'#2'~is~unknown~for~bitset~\tl_to_str:n {#1} }
\prop_gput:Nnn \g_msg_module_name_prop { bitset } { LaTeX }
\prop_gput:Nnn \g_msg_module_type_prop { bitset } { }
```

⟨/package⟩

Chapter 85

l3cctab implementation

```
⟨*package⟩
⟨@@=cctab⟩
```

As LuaTeX offers engine support for category code tables, and this is entirely lacking from the other engines, we need two complementary approaches. (Some future XeTeX may add support, at which point the conditionals below would be different.)

85.1 Variables

\g__cctab_stack_seq
\g__cctab_unused_seq

List of catcode tables saved by nested `\cctab_begin:N`, to restore catcodes at the matching `\cctab_end:`. When popped from the `\g__cctab_stack_seq` the table numbers are stored in `\g__cctab_unused_seq` for later reuse.

```
\seq_new:N \g__cctab_stack_seq
\seq_new:N \g__cctab_unused_seq
```

(*End of definition for* `\g__cctab_stack_seq` *and* `\g__cctab_unused_seq`*.*)

\g__cctab_group_seq

A stack to store the group level when a catcode table started.

```
\seq_new:N \g__cctab_group_seq
```

(*End of definition for* `\g__cctab_group_seq`*.*)

\g__cctab_allocate_int

Integer to keep track of what category code table to allocate. In LuaTeX it is only used in format mode to implement `\cctab_new:N`. In other engines it is used to make csnames for dynamic tables.

```
\int_new:N  \g__cctab_allocate_int
```

(*End of definition for* `\g__cctab_allocate_int`*.*)

\l__cctab_internal_a_tl
\l__cctab_internal_b_tl

Scratch space. For instance, when popping `\g__cctab_stack_seq`/`\g__cctab_unused_seq`, consists of the catcodetable number (integer denotation) in LuaTeX, or of an intarray variable (as a single token) in other engines.

```
\tl_new:N \l__cctab_internal_a_tl
\tl_new:N \l__cctab_internal_b_tl
```

(*End of definition for* `\l__cctab_internal_a_tl` *and* `\l__cctab_internal_b_tl`*.*)

\g__cctab_endlinechar_prop In LuaTeX we store the \endlinechar associated to each \catcodetable in a property list, unless it is the default value 13.

```
\prop_new:N \g__cctab_endlinechar_prop
```

(*End of definition for* \g__cctab_endlinechar_prop.)

85.2 Allocating category code tables

\cctab_new:N
\cctab_new:c
__cctab_new:N
__cctab_gstore:Nnn

The __cctab_new:N auxiliary allocates a new catcode table but does not attempt to set its value consistently across engines. It is used both in \cctab_new:N, which sets catcodes to iniTeX values, and in \cctab_begin:N/\cctab_end: for dynamically allocated tables.

First, the LuaTeX case. Creating a new category code table is done like other registers. In ConTeXt, \newcatcodetable does not include the initialisation, so that is added explicitly.

```
\sys_if_engine_luatex:TF
  {
    \cs_new_protected:Npn \cctab_new:N #1
      {
        \__kernel_chk_if_free_cs:N #1
        \__cctab_new:N #1
      }
    \cs_new_protected:Npn \__cctab_new:N #1
      {
        \newcatcodetable #1
        \tex_initcatcodetable:D #1
      }
  }
```

Now the case for other engines. Here, each table is an integer array. Following the LuaTeX pattern, a new table starts with iniTeX codes. The \debug_suspend: and \debug_resume: functions prevent errors and logging from debug commands which are either duplicate or false when __cctab_new:N is used by \cctab_new:N or \cctab_-const:Nn. The index base is out-by-one, so we have an internal function to handle that. The iniTeX \endlinechar is 13.

```
  {
    \cs_new_protected:Npn \__cctab_new:N #1
      {
        \debug_suspend:
        \intarray_new:Nn #1 { 257 }
        \debug_resume:
      }
    \cs_new_protected:Npn \__cctab_gstore:Nnn #1#2#3
      { \intarray_gset:Nnn #1 { #2 + 1 } {#3} }
    \cs_new_protected:Npn \cctab_new:N #1
      {
        \__kernel_chk_if_free_cs:N #1
        \__cctab_new:N #1
        \int_step_inline:nn { 256 }
          { \__kernel_intarray_gset:Nnn #1 {##1} { 12 } }
        \__kernel_intarray_gset:Nnn #1 { 257 } { 13 }
        \__cctab_gstore:Nnn #1 { 0 } { 9 }
        \__cctab_gstore:Nnn #1 { 13 } { 5 }
```

```
        \__cctab_gstore:Nnn #1 { 32 } { 10 }
        \__cctab_gstore:Nnn #1 { 37 } { 14 }
        \int_step_inline:nnn { 65 } { 90 }
          { \__cctab_gstore:Nnn #1 {##1} { 11 } }
        \__cctab_gstore:Nnn #1 { 92 } { 0 }
        \int_step_inline:nnn { 97 } { 122 }
          { \__cctab_gstore:Nnn #1 {##1} { 11 } }
        \__cctab_gstore:Nnn #1 { 127 } { 15 }
      }
  }
\cs_generate_variant:Nn \cctab_new:N { c }
```

(*End of definition for* `\cctab_new:N`*,* `\__cctab_new:N`*, and* `\__cctab_gstore:Nnn`*. This function is documented on page 288.*)

85.3 Saving category code tables

`\__cctab_gset:n`
`\__cctab_gset_aux:n`

In various functions we need to save the current catcodes (globally) in a table. In LuaTEX, saving the catcodes is a primitives, but the `\endlinechar` needs more work: to avoid filling `\g__cctab_endlinechar_prop` with many entries we special-case the default value 13. In other engines we store 256 current catcodes and the `\endlinechar` in an intarray variable.

```
\sys_if_engine_luatex:TF
  {
    \cs_new_protected:Npn \__cctab_gset:n #1
      { \exp_args:Nf \__cctab_gset_aux:n { \int_eval:n {#1} } }
    \cs_new_protected:Npn \__cctab_gset_aux:n #1
      {
        \tex_savecatcodetable:D #1 \scan_stop:
        \int_compare:nNnTF { \tex_endlinechar:D } = { 13 }
          { \prop_gremove:Nn \g__cctab_endlinechar_prop {#1} }
          {
            \prop_gput:NnV \g__cctab_endlinechar_prop {#1}
              \tex_endlinechar:D
          }
      }
  }
  {
    \cs_new_protected:Npn \__cctab_gset:n #1
      {
        \int_step_inline:nn { 256 }
          {
            \__kernel_intarray_gset:Nnn #1 {##1}
              { \char_value_catcode:n { ##1 - 1 } }
          }
        \__kernel_intarray_gset:Nnn #1 { 257 }
          { \tex_endlinechar:D }
      }
  }
```

(*End of definition for* `\__cctab_gset:n` *and* `\__cctab_gset_aux:n`*.*)

\cctab_gset:Nn
\cctab_gset:cn

Category code tables are always global, so only one version of assignments is needed. Simply run the setup in a group and save the result in a category code table #1, provided it is valid. The internal function is defined above depending on the engine.

```
\cs_new_protected:Npn \cctab_gset:Nn #1#2
  {
    \__cctab_chk_if_valid:NT #1
      {
        \group_begin:
          \cctab_select:N \c_initex_cctab
          #2 \scan_stop:
          \__cctab_gset:n {#1}
        \group_end:
      }
  }
\cs_generate_variant:Nn \cctab_gset:Nn { c }
```

(*End of definition for* `\cctab_gset:Nn`. *This function is documented on page* 288.)

\cctab_gsave_current:N
\cctab_gsave_current:c

Very simple.

```
\cs_new_protected:Npn \cctab_gsave_current:N #1
  {
    \__cctab_chk_if_valid:NT #1
      { \__cctab_gset:n {#1} }
  }
\cs_generate_variant:Nn \cctab_gsave_current:N { c }
```

(*End of definition for* `\cctab_gsave_current:N`. *This function is documented on page* 288.)

85.4 Using category code tables

\g__cctab_internal_cctab
__cctab_internal_cctab_name:

In LuaTeX, we must ensure that the saved tables are read-only. This is done by applying the saved table, then switching immediately to a scratch table. Any later catcode assignment will affect that scratch table rather than the saved one. If we simply switched to the saved tables, then `\char_set_catcode_other:N` in the example below would change `\c_document_cctab` and a later use of that table would give the wrong category code to `_`.

```
\use:n
  {
    \cctab_begin:N \c_document_cctab
      \char_set_catcode_other:N \_
    \cctab_end:
    \cctab_begin:N \c_document_cctab
      \int_compare:nTF { \char_value_catcode:n { `_ } = 8 }
        { \TRUE } { \ERROR }
    \cctab_end:
  }
```

We must also make sure that a scratch table is never reused in a nested group: in the following example, the scratch table used by the first `\cctab_begin:N` would be changed globally by the second one issuing `\savecatcodetable`, and after `\group_end:` the wrong category codes (those of \c_str_cctab) would be imposed. Note that the inner \cctab_-end: restores the correct catcodes only locally, so the problem really comes up because of the different grouping level. The simplest is to use a scratch table labeled by the \currentgrouplevel. We initialize one of them as an example.

```
\use:n
  {
    \cctab_begin:N \c_document_cctab
      \group_begin:
        \cctab_begin:N \c_str_cctab
        \cctab_end:
      \group_end:
    \cctab_end:
  }
```

```
\sys_if_engine_luatex:T
  {
    \__cctab_new:N \g__cctab_internal_cctab
    \cs_new:Npn \__cctab_internal_cctab_name:
      {
        g__cctab_internal
        \tex_romannumeral:D \tex_currentgrouplevel:D
        _cctab
      }
  }
```

(*End of definition for* \g__cctab_internal_cctab *and* __cctab_internal_cctab_name:.)

\cctab_select:N
\cctab_select:c
__cctab_select:N

The public function simply checks the ⟨*cctab var*⟩ exists before using the engine-dependent __cctab_select:N. Skipping these checks would result in low-level engine-dependent errors. First, the LuaTeX case. In other engines, selecting a catcode table is a matter of doing 256 catcode assignments and setting the \endlinechar.

```
\cs_new_protected:Npn \cctab_select:N #1
  { \__cctab_chk_if_valid:NT #1 { \__cctab_select:N #1 } }
\cs_generate_variant:Nn \cctab_select:N { c }
\sys_if_engine_luatex:TF
  {
    \cs_new_protected:Npn \__cctab_select:N #1
      {
        \tex_catcodetable:D #1
        \prop_get:NVNTF \g__cctab_endlinechar_prop #1 \l__cctab_internal_a_tl
          { \int_set:Nn \tex_endlinechar:D { \l__cctab_internal_a_tl } }
          { \int_set:Nn \tex_endlinechar:D { 13 } }
        \cs_if_exist:cF { \__cctab_internal_cctab_name: }
          { \exp_args:Nc \__cctab_new:N { \__cctab_internal_cctab_name: } }
        \exp_args:Nc \tex_savecatcodetable:D { \__cctab_internal_cctab_name: }
        \exp_args:Nc \tex_catcodetable:D { \__cctab_internal_cctab_name: }
      }
  }
  {
    \cs_new_protected:Npn \__cctab_select:N #1
      {
        \int_step_inline:nn { 256 }
          {
```

```
          \char_set_catcode:nn { ##1 - 1 }
            { \__kernel_intarray_item:Nn #1 {##1} }
        }
      \int_set:Nn \tex_endlinechar:D
        { \__kernel_intarray_item:Nn #1 { 257 } }
    }
  }
```

(*End of definition for* `\cctab_select:N` *and* `\__cctab_select:N`*. This function is documented on page 289.*)

`\g__cctab_next_cctab`
`\__cctab_begin_aux:`

For `\cctab_begin:N`/`\cctab_end:` we will need to allocate dynamic tables. This is done here by `\__cctab_begin_aux:`, which puts a table number (in LuaTEX) or name (in other engines) into `\l__cctab_internal_a_tl`. In LuaTEX this simply calls `\__cctab_new:N` and uses the resulting catcodetable number; in other engines we need to give a name to the intarray variable and use that. In LuaTEX, to restore catcodes at `\cctab_end:` we cannot just set `\catcodetable` to its value before `\cctab_begin:N`, because that table may have been altered by other code in the mean time. So we must make sure to save the catcodes in a table we control and restore them at `\cctab_end:`.

```
\sys_if_engine_luatex:TF
  {
    \cs_new_protected:Npn \__cctab_begin_aux:
      {
        \__cctab_new:N \g__cctab_next_cctab
        \tl_set:NV \l__cctab_internal_a_tl \g__cctab_next_cctab
        \cs_undefine:N \g__cctab_next_cctab
      }
  }
  {
    \cs_new_protected:Npn \__cctab_begin_aux:
      {
        \int_gincr:N \g__cctab_allocate_int
        \exp_args:Nc \__cctab_new:N
          { g__cctab_ \int_use:N \g__cctab_allocate_int _cctab }
        \exp_args:NNc \tl_set:Nn \l__cctab_internal_a_tl
          { g__cctab_ \int_use:N \g__cctab_allocate_int _cctab }
      }
  }
```

(*End of definition for* `\g__cctab_next_cctab` *and* `\__cctab_begin_aux:`*.*)

`\cctab_begin:N`
`\cctab_begin:c`

Check the ⟨*cctab var*⟩ exists, to avoid low-level errors. Get in `\l__cctab_internal_a_tl` the number/name of a dynamic table, either from `\g__cctab_unused_seq` where we save tables that are not currently in use, or from `\__cctab_begin_aux:` if none are available. Then save the current catcodes into the table (pointed to by) `\l__cctab_internal_a_tl` and save that table number in a stack before selecting the desired catcodes.

```
\cs_new_protected:Npn \cctab_begin:N #1
  {
    \__cctab_chk_if_valid:NT #1
      {
        \seq_gpop:NNF \g__cctab_unused_seq \l__cctab_internal_a_tl
          { \__cctab_begin_aux: }
        \__cctab_chk_group_begin:e
```

```
        { \__cctab_nesting_number:N \l__cctab_internal_a_tl }
      \seq_gpush:NV \g__cctab_stack_seq \l__cctab_internal_a_tl
      \exp_args:NV \__cctab_gset:n \l__cctab_internal_a_tl
      \__cctab_select:N #1
    }
  }
\cs_generate_variant:Nn \cctab_begin:N { c }
```

(*End of definition for* `\cctab_begin:N`. *This function is documented on page 289.*)

`\cctab_end:` Make sure a `\cctab_begin:N` was used some time earlier, get in `\l__cctab_internal_a_tl` the catcode table number/name in which the prevailing catcodes were stored, then restore these catcodes. The dynamic table is now unused hence stored in `\g__cctab_unused_seq` for recycling by later `\cctab_begin:N`.

```
\cs_new_protected:Npn \cctab_end:
  {
    \seq_gpop:NNTF \g__cctab_stack_seq \l__cctab_internal_a_tl
      {
        \seq_gpush:NV \g__cctab_unused_seq \l__cctab_internal_a_tl
        \exp_args:Ne \__cctab_chk_group_end:n
          { \__cctab_nesting_number:N \l__cctab_internal_a_tl }
        \__cctab_select:N \l__cctab_internal_a_tl
      }
      { \msg_error:nn { cctab } { extra-end } }
  }
```

(*End of definition for* `\cctab_end:`. *This function is documented on page 289.*)

`\__cctab_chk_group_begin:n`
`\__cctab_chk_group_begin:e`
`\__cctab_chk_group_end:n`

Catcode tables are not allowed to be intermixed with groups, so here we check that they are properly nested regarding TeX groups. `\__cctab_chk_group_begin:n` stores the current group level in a stack, and locally defines a dummy control sequence `\__cctab_group_`⟨*cctab-level*⟩`_chk:`.

`\__cctab_chk_group_end:n` pops the stack, and compares the returned value with `\tex_currentgrouplevel:D`. If they differ, `\cctab_end:` is in a different grouping level than the matching `\cctab_begin:N`. If they are the same, both happened at the same level, however a group might have ended and another started between `\cctab_begin:N` and `\cctab_end:`:

```
\group_begin:
  \cctab_begin:N \c_document_cctab
\group_end:
\group_begin:
  \cctab_end:
\group_end:
```

In this case checking `\tex_currentgrouplevel:D` is not enough, so we locally define `\__cctab_group_`⟨*cctab-level*⟩`_chk:`, and then check if it exist in `\cctab_end:`. If it doesn't, we know there was a group end where it shouldn't.

The ⟨*cctab-level*⟩ in the sentinel macro above cannot be replaced by the more convenient `\tex_currentgrouplevel:D` because with the latter we might be tricked. Suppose:

```
\group_begin:
  \cctab_begin:N \c_code_cctab % A
\group_end:
\group_begin:
  \cctab_begin:N \c_code_cctab % B
  \cctab_end: % C
  \cctab_end: % D
\group_end:
```

The line marked with `A` would start a `cctab` with a sentinel token named `\__cctab_-group_1_chk:`, which would disappear at the `\group_end:` that follows. But `B` would create the same sentinel token, since both are at the same group level. Line `C` would end the `cctab` from line `B` correctly, but so would line `D` because line `B` created the same sentinel token. Using ⟨*cctab-level*⟩ works correctly because it signals that certain `cctab` level was activated somewhere, but if it doesn't exist when the `\cctab_end:` is reached, we had a problem.

Unfortunately these tests only flag the wrong usage at the `\cctab_end:`, which might be far from the `\cctab_begin:N`. However it isn't possible to signal the wrong usage at the `\group_end:` without using `\tex_aftergroup:D`, which is unsafe in certain types of groups.

The three cases checked here just raise an error, and no recovery is attempted: usually interleaving groups and catcode tables will work predictably.

```
\cs_new_protected:Npn \__cctab_chk_group_begin:n #1
  {
    \seq_gpush:Ne \g__cctab_group_seq
      { \int_use:N \tex_currentgrouplevel:D }
    \cs_set_eq:cN { __cctab_group_ #1 _chk: } \prg_do_nothing:
  }
\cs_generate_variant:Nn \__cctab_chk_group_begin:n { e }
\cs_new_protected:Npn \__cctab_chk_group_end:n #1
  {
    \seq_gpop:NN \g__cctab_group_seq \l__cctab_internal_b_tl
    \bool_lazy_and:nnF
      {
        \int_compare_p:nNn
          { \tex_currentgrouplevel:D } = { \l__cctab_internal_b_tl }
      }
      { \cs_if_exist_p:c { __cctab_group_ #1 _chk: } }
      {
        \msg_error:nne { cctab } { group-mismatch }
          {
            \int_sign:n
              { \tex_currentgrouplevel:D - \l__cctab_internal_b_tl }
          }
      }
    \cs_undefine:c { __cctab_group_ #1 _chk: }
  }
```

(*End of definition for* `\__cctab_chk_group_begin:n` *and* `\__cctab_chk_group_end:n`*.*)

`\__cctab_nesting_number:N`
`\__cctab_nesting_number:w`

This macro returns the numeric index of the current catcode table. In LuaTeX this is just the argument, which is a count reference to a `\catcodetable` register. In other engines, the number is extracted from the `cctab` variable.

```
\sys_if_engine_luatex:TF
  { \cs_new:Npn \__cctab_nesting_number:N #1 {#1} }
  {
    \cs_new:Npn \__cctab_nesting_number:N #1
      {
        \exp_after:wN \exp_after:wN \exp_after:wN \__cctab_nesting_number:w
          \exp_after:wN \token_to_str:N #1
      }
    \use:e
      {
        \cs_new:Npn \exp_not:N \__cctab_nesting_number:w
          #1 \tl_to_str:n { g__cctab_ } #2 \tl_to_str:n { _cctab } {#2}
      }
  }
```

(*End of definition for* `\__cctab_nesting_number:N` *and* `\__cctab_nesting_number:w`.)

Finally, install some code at the end of the TeX run to check that all `\cctab_begin:N` were ended by some `\cctab_end:`.

```
\cs_if_exist:NT \hook_gput_code:nnn
  {
    \hook_gput_code:nnn { enddocument/end } { cctab }
      {
        \seq_if_empty:NF \g__cctab_stack_seq
          { \msg_error:nn { cctab } { missing-end } }
      }
  }
```

`\cctab_item:Nn`
`\cctab_item:cn`

Evaluate the integer argument only once. In most engines the `cctab` variable only has 256 entries so we only look up the catcode for these entries, otherwise we use the current catcode. In particular, for out-of-range values we use whatever fall-back `\char_value_catcode:n`. In LuaTeX, we use the `tex.getcatcode` function.

```
\cs_new:Npn \cctab_item:Nn #1#2
  { \exp_args:Nf \__cctab_item:nN { \int_eval:n {#2} } #1 }
\sys_if_engine_luatex:TF
  {
    \cs_new:Npn \__cctab_item:nN #1#2
      { \lua_now:e { tex.print(-2, tex.getcatcode(\int_use:N #2, #1)) } }
  }
  {
    \cs_new:Npn \__cctab_item:nN #1#2
      {
        \int_compare:nNnTF {#1} < { 256 }
          { \intarray_item:Nn #2 { #1 + 1 } }
          { \char_value_catcode:n {#1} }
      }
  }
\cs_generate_variant:Nn \cctab_item:Nn { c }
```

(*End of definition for* `\cctab_item:Nn`. *This function is documented on page 289.*)

85.5 Category code table conditionals

`\cctab_if_exist_p:N`
`\cctab_if_exist_p:c`
`\cctab_if_exist:NTF`
`\cctab_if_exist:cTF`

Checks whether a ⟨*cctab var*⟩ is defined.

```
\prg_new_eq_conditional:NNn \cctab_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \cctab_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

(*End of definition for* `\cctab_if_exist:NTF`. *This function is documented on page* 289.)

`\__cctab_chk_if_valid:NTF`
`\__cctab_chk_if_valid_aux:NTF`

Checks whether the argument is defined and whether it is a valid ⟨*cctab var*⟩. In LuaTeX the validity of the ⟨*cctab var*⟩ is checked by the engine, which complains if the argument is not a `\chardef`'ed constant. In other engines, check if the given command is an intarray variable (the underlying definition is a copy of the `cmr10` font).

```
\prg_new_protected_conditional:Npnn \__cctab_chk_if_valid:N #1
  { TF , T , F }
  {
    \cctab_if_exist:NTF #1
      {
        \__cctab_chk_if_valid_aux:NTF #1
          { \prg_return_true: }
          {
            \msg_error:nne { cctab } { invalid-cctab }
              { \token_to_str:N #1 }
            \prg_return_false:
          }
      }
      {
        \msg_error:nne { kernel } { command-not-defined }
          { \token_to_str:N #1 }
        \prg_return_false:
      }
  }
\sys_if_engine_luatex:TF
  {
    \cs_new_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
      {
        \int_compare:nNnTF {#1-1} < { \e@alloc@ccodetable@count }
      }
    \cs_if_exist:NT \c_syst_catcodes_n
      {
        \cs_gset_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
          {
            \int_compare:nTF { #1 <= \c_syst_catcodes_n }
          }
      }
  }
  {
    \cs_new_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
      {
        \exp_args:Nf \str_if_in:nnTF
          { \cs_meaning:N #1 }
          { select~font~cmr10~at~ }
      }
  }
```

(*End of definition for* `\__cctab_chk_if_valid:NTF` *and* `\__cctab_chk_if_valid_aux:NTF`.)

85.6 Constant category code tables

\cctab_const:Nn
\cctab_const:cn

Creates a new ⟨*cctab var*⟩ then sets it with the iniTEX and user-supplied codes. To avoid false debug errors, we write out implementation of \cctab_new:N and \cctab_gset:Nn instead of directly using them here. The initialization part in \cctab_new:N in non-LuaTEX is omitted as it's covered by the iniTEX settings.

```
\cs_new_protected:Npn \cctab_const:Nn #1#2
  {
    \__kernel_chk_if_free_cs:N #1
    \__cctab_new:N #1
    \group_begin:
      \cctab_select:N \c_initex_cctab
      #2 \scan_stop:
      \__cctab_gset:n {#1}
    \group_end:
  }
\cs_generate_variant:Nn \cctab_const:Nn { c }
```

(*End of definition for* \cctab_const:Nn. *This function is documented on page 288.*)

\c_initex_cctab
\c_other_cctab
\c_str_cctab

Creating category code tables means thinking starting from iniTEX. For all-other and the standard "string" tables that's easy.

```
\cctab_new:N \c_initex_cctab
\cctab_const:Nn \c_other_cctab
  {
    \cctab_select:N \c_initex_cctab
    \int_set:Nn \tex_endlinechar:D     { -1 }
    \int_step_inline:nnn { 0 } { 127 }
      { \char_set_catcode_other:n {#1} }
  }
\cctab_const:Nn \c_str_cctab
  {
    \cctab_select:N \c_other_cctab
    \char_set_catcode_space:n { 32 }
  }
```

(*End of definition for* \c_initex_cctab*,* \c_other_cctab*, and* \c_str_cctab*. These variables are documented on page 290.*)

\c_code_cctab
\c_document_cctab

To pick up document-level category codes, we need to delay set up to the end of the format, where that's possible. Also, as there are a *lot* of category codes to set, we avoid using the official interface and store the document codes using internal code. Depending on whether we are in the hook or not, the catcodes may be code or document, so we explicitly set up both correctly.

```
\cs_if_exist:NTF \@expl@finalise@setup@@
  { \tl_gput_right:Nn \@expl@finalise@setup@@ }
  { \use:n }
  {
    \__cctab_new:N \c_code_cctab
    \group_begin:
      \int_set:Nn \tex_endlinechar:D { 32 }
      \char_set_catcode_invalid:n { 0 }
      \bool_lazy_or:nnTF
        { \sys_if_engine_xetex_p: } { \sys_if_engine_luatex_p: }
```

```
      { \int_step_function:nN { 31 } \char_set_catcode_invalid:n }
      { \int_step_function:nN { 31 } \char_set_catcode_active:n }
    \int_step_function:nnN { 33 } { 64 } \char_set_catcode_other:n
    \int_step_function:nnN { 65 } { 90 } \char_set_catcode_letter:n
    \int_step_function:nnN { 91 } { 96 } \char_set_catcode_other:n
    \int_step_function:nnN { 97 } { 122 } \char_set_catcode_letter:n
    \char_set_catcode_ignore:n           { 9 }   % tab
    \char_set_catcode_other:n            { 10 }  % lf
    \char_set_catcode_active:n           { 12 }  % ff
    \char_set_catcode_end_line:n         { 13 }  % cr
    \char_set_catcode_ignore:n           { 32 }  % space
    \char_set_catcode_parameter:n        { 35 }  % hash
    \char_set_catcode_math_toggle:n      { 36 }  % dollar
    \char_set_catcode_comment:n          { 37 }  % percent
    \char_set_catcode_alignment:n        { 38 }  % ampersand
    \char_set_catcode_letter:n           { 58 }  % colon
    \char_set_catcode_escape:n           { 92 }  % backslash
    \char_set_catcode_math_superscript:n { 94 }  % circumflex
    \char_set_catcode_letter:n           { 95 }  % underscore
    \char_set_catcode_group_begin:n      { 123 } % left brace
    \char_set_catcode_other:n            { 124 } % pipe
    \char_set_catcode_group_end:n        { 125 } % right brace
    \char_set_catcode_space:n            { 126 } % tilde
    \char_set_catcode_invalid:n          { 127 } % ^^?
    \bool_lazy_or:nnF
      { \sys_if_engine_xetex_p: } { \sys_if_engine_luatex_p: }
      { \int_step_function:nnN { 128 } { 255 } \char_set_catcode_active:n }
    \__cctab_gset:n { \c_code_cctab }
  \group_end:
  \cctab_const:Nn \c_document_cctab
    {
      \cctab_select:N \c_code_cctab
      \int_set:Nn \tex_endlinechar:D { 13 }
      \char_set_catcode_space:n          { 9 }
      \char_set_catcode_space:n          { 32 }
      \char_set_catcode_other:n          { 58 }
      \char_set_catcode_math_subscript:n { 95 }
      \char_set_catcode_active:n         { 126 }
    }
}
```

(*End of definition for* `\c_code_cctab` *and* `\c_document_cctab`*. These variables are documented on page* *289*.)

\g_tmpa_cctab
\g_tmpb_cctab

```
\cctab_new:N \g_tmpa_cctab
\cctab_new:N \g_tmpb_cctab
```

(*End of definition for* `\g_tmpa_cctab` *and* `\g_tmpb_cctab`*. These variables are documented on page* *290*.)

85.7 Messages

```
\msg_new:nnnn { cctab } { stack-full }
  { The~category~code~table~stack~is~exhausted. }
```

```
  {
    LaTeX~has~been~asked~to~switch~to~a~new~category~code~table,~
    but~there~is~no~more~space~to~do~this!
  }
\msg_new:nnnn { cctab } { extra-end }
  { Extra~\iow_char:N\\cctab_end:~ignored~\msg_line_context:. }
  {
    LaTeX~came~across~a~\iow_char:N\\cctab_end:~without~a~matching~
    \iow_char:N\\cctab_begin:N.~This~command~will~be~ignored.
  }
\msg_new:nnnn { cctab } { missing-end }
  { Missing~\iow_char:N\\cctab_end:~before~end~of~TeX~run. }
  {
    LaTeX~came~across~more~\iow_char:N\\cctab_begin:N~than~
    \iow_char:N\\cctab_end:.
  }
\msg_new:nnnn { cctab } { invalid-cctab }
  { Invalid~\iow_char:N\\catcode~table. }
  {
    You~can~only~switch~to~a~\iow_char:N\\catcode~table~that~is~
    initialized~using~\iow_char:N\\cctab_new:N~or~
    \iow_char:N\\cctab_const:Nn.
  }
\msg_new:nnnn { cctab } { group-mismatch }
  {
    \iow_char:N\\cctab_end:~occurred~in~a~
    \int_case:nn {#1}
      {
        { 0 } { different~group }
        { 1 } { higher~group~level }
        { -1 } { lower~group~level }
      } ~than~
    the~matching~\iow_char:N\\cctab_begin:N.
  }
  {
    Catcode~tables~and~groups~must~be~properly~nested,~but~
    you~tried~to~interleave~them.~LaTeX~will~try~to~proceed,~
    but~results~may~be~unexpected.
  }
\prop_gput:Nnn \g_msg_module_name_prop { cctab } { LaTeX }
\prop_gput:Nnn \g_msg_module_type_prop { cctab } { }
```
⟨/package⟩

Chapter 86

l3unicode implementation

```
⟨*package⟩
⟨@@=codepoint⟩
```

86.1 User functions

\codepoint_str_generate:n
__codepoint_str_generate:nnnn
\codepoint_generate:nn
__codepoint_generate:nnnn
__codepoint_generate:n

Conversion of a codepoint to a character (Unicode engines) or to one or more bytes (8-bit engines) is required. For loading the data, all that is needed is the form which creates strings: these are outside the group as they will also be used when looking up data in the hash table storage at point-of-use. Later, we will also need functions that can generate character tokens for document use: those are defined below, in the data recovery setup.

```
\bool_lazy_or:nnTF
  { \sys_if_engine_luatex_p: }
  { \sys_if_engine_xetex_p: }
  {
    \cs_new:Npn \codepoint_str_generate:n #1
      {
        \int_compare:nNnTF {#1} = { `\  }
          { ~ }
          { \char_generate:nn {#1} { 12 } }
      }
    \cs_new:Npn \codepoint_generate:nn #1#2
      {
        \int_compare:nNnTF {#1} = { `\  }
          { ~ }
          {
            \__kernel_exp_not:w \exp_after:wN \exp_after:wN \exp_after:wN
              { \char_generate:nn {#1} {#2} }
          }
      }
  }
  {
    \cs_new:Npn \codepoint_str_generate:n #1
      {
        \int_compare:nNnTF {#1} = { `\  }
          { ~ }
          {
            \use:e
```

```
          {
            \exp_not:N \__codepoint_str_generate:nnnn
              \__kernel_codepoint_to_bytes:n {#1}
          }
      }
  }
\cs_new:Npn \__codepoint_str_generate:nnnn #1#2#3#4
  {
    \char_generate:nn {#1} { 12 }
    \tl_if_blank:nF {#2}
      {
        \char_generate:nn {#2} { 12 }
        \tl_if_blank:nF {#3}
          {
            \char_generate:nn {#3} { 12 }
            \tl_if_blank:nF {#4}
              { \char_generate:nn {#4} { 12 } }
          }
      }
  }
\cs_new:Npn \codepoint_generate:nn #1#2
  {
    \int_compare:nNnTF {#1} = { `\  }
      { ~ }
      {
        \int_compare:nNnTF {#1} < { "80 }
          {
            \__kernel_exp_not:w \exp_after:wN \exp_after:wN \exp_after:wN
              { \char_generate:nn {#1} {#2} }
          }
          {
            \use:e
              {
                \exp_not:N \__codepoint_generate:nnnn
                  \__kernel_codepoint_to_bytes:n {#1}
              }
          }
      }
  }
\cs_new:Npn \__codepoint_generate:nnnn #1#2#3#4
  {
    \__kernel_exp_not:w \exp_after:wN
      {
        \tex_expanded:D
          {
            \__codepoint_generate:n {#1}
            \__codepoint_generate:n {#2}
            \tl_if_blank:nF {#3}
              {
                \__codepoint_generate:n {#3}
                \tl_if_blank:nF {#4}
                  { \__codepoint_generate:n {#4} }
              }
          }
```

```
          }
      }
    \cs_new:Npn \__codepoint_generate:n #1
      {
        \__kernel_exp_not:w \exp_after:wN \exp_after:wN \exp_after:wN
          { \char_generate:nn {#1} { 13 } }
      }
  }
```

(*End of definition for* `\codepoint_str_generate:n` *and others. These functions are documented on page 293.*)

__kernel_codepoint_to_bytes:n
__codepoint_to_bytes_auxi:n
__codepoint_to_bytes_auxii:Nnn
__codepoint_to_bytes_auxiii:n
__codepoint_to_bytes_outputi:nw
__codepoint_to_bytes_outputii:nw
__codepoint_to_bytes_outputiii:nw
__codepoint_to_bytes_outputiv:nw
__codepoint_to_bytes_output:nnn
__codepoint_to_bytes_output:fnn
__codepoint_to_bytes_end:

This code converts a codepoint into the correct UTF-8 representation. In terms of the algorithm itself, see https://en.wikipedia.org/wiki/UTF-8 for the octet pattern.

```
\cs_new:Npn \__kernel_codepoint_to_bytes:n #1
  {
    \exp_args:Nf \__codepoint_to_bytes_auxi:n
      { \int_eval:n {#1} }
  }
\cs_new:Npn \__codepoint_to_bytes_auxi:n #1
  {
    \if_int_compare:w #1 > "80 \exp_stop_f:
      \if_int_compare:w #1 < "800 \exp_stop_f:
        \__codepoint_to_bytes_outputi:nw
          { \__codepoint_to_bytes_auxii:Nnn C {#1} { 64 } }
        \__codepoint_to_bytes_outputii:nw
          { \__codepoint_to_bytes_auxiii:n {#1} }
      \else:
        \if_int_compare:w #1 < "10000 \exp_stop_f:
          \__codepoint_to_bytes_outputi:nw
            { \__codepoint_to_bytes_auxii:Nnn E {#1} { 64 * 64 } }
          \__codepoint_to_bytes_outputii:nw
            {
              \__codepoint_to_bytes_auxiii:n
                { \int_div_truncate:nn {#1} { 64 } }
            }
          \__codepoint_to_bytes_outputiii:nw
            { \__codepoint_to_bytes_auxiii:n {#1} }
        \else:
          \__codepoint_to_bytes_outputi:nw
            {
              \__codepoint_to_bytes_auxii:Nnn F
                {#1} { 64 * 64 * 64 }
            }
          \__codepoint_to_bytes_outputii:nw
            {
              \__codepoint_to_bytes_auxiii:n
                { \int_div_truncate:nn {#1} { 64 * 64 } }
            }
          \__codepoint_to_bytes_outputiii:nw
            {
              \__codepoint_to_bytes_auxiii:n
                { \int_div_truncate:nn {#1} { 64 } }
            }
```

```
          \__codepoint_to_bytes_outputiv:nw
            { \__codepoint_to_bytes_auxiii:n {#1} }
        \fi:
      \fi:
    \else:
      \__codepoint_to_bytes_outputi:nw {#1}
    \fi:
    \__codepoint_to_bytes_end: { } { } { } { }
  }
\cs_new:Npn \__codepoint_to_bytes_auxii:Nnn #1#2#3
  {  "#10 + \int_div_truncate:nn {#2} {#3} }
\cs_new:Npn \__codepoint_to_bytes_auxiii:n #1
  { \int_mod:nn {#1} { 64 } + 128 }
\cs_new:Npn \__codepoint_to_bytes_outputi:nw
  #1 #2 \__codepoint_to_bytes_end: #3
  { \__codepoint_to_bytes_output:fnn { \int_eval:n {#1} } { } {#2} }
\cs_new:Npn \__codepoint_to_bytes_outputii:nw
  #1 #2 \__codepoint_to_bytes_end: #3#4
  { \__codepoint_to_bytes_output:fnn { \int_eval:n {#1} } { {#3} } {#2} }
\cs_new:Npn \__codepoint_to_bytes_outputiii:nw
  #1 #2 \__codepoint_to_bytes_end: #3#4#5
  {
    \__codepoint_to_bytes_output:fnn
      { \int_eval:n {#1} } { {#3} {#4} } {#2}
  }
\cs_new:Npn \__codepoint_to_bytes_outputiv:nw
  #1 #2 \__codepoint_to_bytes_end: #3#4#5#6
  {
    \__codepoint_to_bytes_output:fnn
      { \int_eval:n {#1} } { {#3} {#4} {#5} } {#2}
  }
\cs_new:Npn \__codepoint_to_bytes_output:nnn #1#2#3
  {
    #3
    \__codepoint_to_bytes_end: #2 {#1}
  }
\cs_generate_variant:Nn \__codepoint_to_bytes_output:nnn { f }
\cs_new:Npn \__codepoint_to_bytes_end: { }
```

(*End of definition for* `\__kernel_codepoint_to_bytes:n` *and others.*)

`\codepoint_to_category:n` Get the value and convert back to the string.

```
\cs_new:Npn \codepoint_to_category:n #1
  {
    \cs:w
      c__codepoint_category_
      \tex_romannumeral:D
        \__kernel_codepoint_data:nn { category } {#1}
      _str
    \cs_end:
  }
```

(*End of definition for* `\codepoint_to_category:n`. *This function is documented on page 294.*)

\codepoint_to_nfd:n
__codepoint_to_nfd:n
__codepoint_to_nfd:nn
__codepoint_to_nfd:nnn
__codepoint_to_nfd:nnnn

Converted to NFD is a potentially-recursive process: the key is to check if we get the input codepoint back again. As far as possible, we use the same path for all engines.

```
\cs_new:Npn \codepoint_to_nfd:n #1
  { \exp_args:Ne \__codepoint_to_nfd:n { \int_eval:n {#1} } }
\cs_new:Npn \__codepoint_to_nfd:n #1
  { \__codepoint_to_nfd:nn {#1} { \char_value_catcode:n {#1} } }
\bool_lazy_or:nnF
  { \sys_if_engine_luatex_p: }
  { \sys_if_engine_xetex_p: }
  {
    \cs_gset:Npn \__codepoint_to_nfd:n #1
      {
        \int_compare:nNnTF {#1} > { "80 }
          { \__codepoint_to_nfd:nn {#1} { 12 } }
          { \__codepoint_to_nfd:nn {#1} { \char_value_catcode:n {#1} } }
      }
  }
\cs_new:Npn \__codepoint_to_nfd:nn #1#2
  {
    \exp_args:Ne \__codepoint_to_nfd:nnn
      { \__codepoint_nfd:n {#1} } {#1} {#2}
  }
\cs_new:Npn \__codepoint_to_nfd:nnn #1#2#3 { \__codepoint_to_nfd:nnnn #1 {#2} {#3} }
\cs_new:Npn \__codepoint_to_nfd:nnnn #1#2#3#4
  {
    \int_compare:nNnTF {#1} = {#3}
      { \codepoint_generate:nn {#1} {#4} }
      {
        \__codepoint_to_nfd:nn {#1} {#4}
        \tl_if_blank:nF {#2}
          { \__codepoint_to_nfd:nn {#2} {#4} }
      }
  }
```

(*End of definition for* \codepoint_to_nfd:n *and others. This function is documented on page 294.*)

86.2 Data loader

Text operations requires data from the Unicode Consortium. Data read into Unicode engine formats is at best a small part of what we need, so there is a loader here to set up the appropriate data structures.

Where we need data for most or all of the Unicode range, we use the two-stage table approach recommended by the Unicode Consortium and demonstrated in a model implementation in Python in https://www.strchr.com/multi-stage_tables. This approach uses the intarray (fontdimen-based) data type as it is fast for random access and avoids significant hash table usage. In contrast, where only a small subset of codepoints are required, storage as macros is preferable. There is also some consideration of the effort needed to load data: see for example the grapheme breaking information, which would be problematic to convert into a two-stage table but which can be used with reasonable performance in a small number of comma lists (at the cost that breaking at higher codepoint Hangul characters will be slightly slow).

\c__codepoint_block_size_int Choosing the block size for the blocks in the two-stage approach is non-trivial: depending on the data stored, the optimal size for memory usage will vary. At the same time, for us there is also the question of load-time: larger blocks require longer comma lists as intermediates, so are slower. As this is going to be needed to use the data, we set it up outside of the group for clarity.

```
\int_const:Nn \c__codepoint_block_size_int { 64 }
```

(*End of definition for* `\c__codepoint_block_size_int`.)

Parsing the data files can be the same way for all engines, but where they are stored as character tokens, the construction method depends on whether they are Unicode or 8-bit internally. Parsing is therefore done by common functions, with some data storage using engine-specific auxiliaries.

As only the data needs to remain at the end of this process, everything is set up inside a group. The only thing that is outside is creating a stream: they are global anyway and it is best to force a stream for all engines.

\g__codepoint_data_ior

```
\ior_new:N \g__codepoint_data_ior
```

(*End of definition for* `\g__codepoint_data_ior`.)

We need some setup for the two-part table approach. The number of blocks we need will be variable, but the resulting size of the stage one table is predictable. For performance reasons, we therefore create the stage one tables now so they can be used immediately, and will later rename them as a constant tables. For each two-stage table construction, we need a comma list to hold the partial block and a couple of integers to track where we are up to. To avoid burning registers, the latter are stored in macros and are "fake" integers. We also avoid any `new` functions, keeping as much as possible local.

As we need both positive and negative values, case data requires one two-stage table for each transformation. In contrasts, general Unicode properties could be stored in one table with appropriate combination rules: that is not done at present but is likely to be added over time. Here, all that is needed is additional entries into the comma-list to create the structures.

Notice that in the standard expl3 way we are indexes position not offset: that does mean a little work later.

```
\group_begin:
  \clist_map_inline:nn
    { category , uppercase , lowercase }
    {
      \cs_set_nopar:cpn { l__codepoint_ #1 _block_clist } { }
      \cs_set_nopar:cpn { l__codepoint_ #1 _block_tl } { 1 }
      \cs_set_nopar:cpn { l__codepoint_ #1 _pos_tl } { 0 }
      \intarray_new:cn { g__codepoint_ #1 _index_intarray }
        { \int_div_truncate:nn { "110000 } \c__codepoint_block_size_int }
    }
```

We need an integer value when matching the current block to those we have already seen, and a way to track codepoints for handling ranges. Again, we avoid using up registers or creating global names.

```
  \cs_set_nopar:Npn \l__codepoint_next_codepoint_fint_tl { 0 }
  \cs_set_nopar:Npn \l__codepoint_matched_block_tl { 0 }
```

For Unicode general category, there needs to be numerical representation of each possible value. As we need to go from string to number here, but the other way elsewhere, we set up fast mappings both ways, but one set local and the other as constants.

```
\cs_set_protected:Npn \__codepoint_data_auxi:w #1#2
  {
    \quark_if_recursion_tail_stop:n {#2}
    \cs_set_nopar:cpn { l__codepoint_category_ #2 _tl } {#1}
    \str_const:cn { c__codepoint_category_ \tex_romannumeral:D #1 _str } {#2}
    \exp_args:Ne \__codepoint_data_auxi:w { \int_eval:n { #1 + 1 } }
  }
\__codepoint_data_auxi:w { 1 }
  { Lu } { Ll } { Lt } { Lm } { Lo }
  { Mn } { Me } { Mc }
  { Nd } { Nl } { No }
  { Zs } { Zl } { Zp }
  { Cc } { Cf } { Co } { Cs } { Cn }
  { Pd } { Ps } { Pe } { Pc } { Po } { Pi } { Pf }
  { Sm } { Sc } { Sk } { So }
  \q_recursion_tail
  \q_recursion_stop
```

Parse the main Unicode data file and pull out the NFD and case changing data. The NFD data is stored on using the hash table approach and can yield a predictable number of codepoints: one or two. We also need the case data, which will be modified further below. To allow for finding ranges, the description of the codepoint needs to be carried forward.

```
\cs_set_protected:Npn \__codepoint_data_auxi:w
  #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ;
  {
    \tl_if_blank:nF {#6}
      {
        \tl_if_head_eq_charcode:nNF {#6}  < % >
          { \__codepoint_data_auxii:w #1 ; #6 ~ \q_stop }
      }
    \__codepoint_data_auxiii:w #1 ; #2 ; #3 ;
  }
\cs_set_protected:Npn \__codepoint_data_auxii:w #1 ; #2 ~ #3 \q_stop
  {
    \tl_const:ce
      { c__codepoint_nfd_ \codepoint_str_generate:n {"#1} _tl }
      {
        {"#2}
        { \tl_if_blank:nF {#3} {"#3} }
      }
  }
```

The category data needs to be converted from a string to the numerical equivalent: a simple operation. The case data is going to be stored as an offset from the parent character, rather than an absolute value. We therefore deal with that plus the situation where a codepoint has no mapping data in one shot.

```
\cs_set_protected:Npn \__codepoint_data_auxiii:w
  #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ~ \q_stop
  {
    \use:e
```

```
        {
          \__codepoint_data_auxiv:w
            #1 ; #2 ;
            \__codepoint_data_category:n {#3} ;
            \__codepoint_data_offset:nn {#1} {#7} ;
            \__codepoint_data_offset:nn {#1} {#8} ;
            #9;
        }
    }
  \cs_set:Npn \__codepoint_data_category:n #1
    { \use:c { l__codepoint_category_ #1 _tl } }
  \cs_set:Npn \__codepoint_data_offset:nn #1#2
    {
      \tl_if_blank:nTF {#2}
        { 0 }
        { \int_eval:n { "#2 - "#1 } }
    }
```

To deal with ranges, we track the position of the next codepoint expected. If there is a gap, we deal with that separately: it could be a range or an unused part of the Unicode space. As such, we deal with the current codepoint here whether or not there is range to fill in. Upper- and lowercase data go into the two-stage table, any titlecase exception is just stored in a macro. The data for the codepoint is added to the current block, and if that is now complete we move on to save the block. The case exceptions are all stored as codepoints, with a fixed number of balanced text as we know that there are never more than three.

```
  \cs_set_protected:Npn \__codepoint_data_auxiv:w #1 ; #2 ; #3 ; #4 ; #5 ; #6 ;
    {
      \int_compare:nNnT {"#1} > \l__codepoint_next_codepoint_fint_tl
        {
          \__codepoint_data_auxv:nnnnw {#1} {#3} {#4} {#5}
            #2 Last> \q_stop
        }
      \__codepoint_add:nn { category } {#3}
      \__codepoint_add:nn { uppercase } {#4}
      \__codepoint_add:nn { lowercase } {#5}
      \int_compare:nNnF {#4} = { \__codepoint_data_offset:nn {#1} {#6} }
        {
          \tl_const:ce
            { c__codepoint_titlecase_ \codepoint_str_generate:n {"#1} _tl }
            { {"#6} { } { } }
        }
      \tl_set:Ne \l__codepoint_next_codepoint_fint_tl
        { \int_eval:n { "#1 + 1 } }
    }
  \cs_set_protected:Npn \__codepoint_add:nn #1#2
    {
      \clist_put_right:cn { l__codepoint_ #1 _block_clist } {#2}
      \int_compare:nNnT { \clist_count:c { l__codepoint_ #1 _block_clist } }
        = \c__codepoint_block_size_int
        { \__codepoint_save_blocks:nn {#1} { 1 } }
    }
```

Distinguish between a range and a gap, and pass on the appropriate value(s). The general category for unassigned characters is Cn, so we find the correct value once and then use that.

```
\cs_set_protected:Npe \__codepoint_data_auxv:nnnnw #1#2#3#4#5 Last> #6 \q_stop
  {
    \exp_not:N \tl_if_blank:nTF {#6}
      {
        \exp_not:N \__codepoint_range:nnn {#1} { category }
          { \exp_not:V \l__codepoint_category_Cn_tl }
        \exp_not:N \__codepoint_range:nnn {#1} { uppercase } { 0 }
        \exp_not:N \__codepoint_range:nnn {#1} { lowercase } { 0 }
      }
      {
        \exp_not:N \__codepoint_range:nnn {#1} { category } {#2}
        \exp_not:N \__codepoint_range:nnn {#1} { uppercase } {#3}
        \exp_not:N \__codepoint_range:nnn {#1} { lowercase } {#4}
      }
  }
```

Calculated the length of the range and the space remaining in the current block.

```
\cs_set_protected:Npn \__codepoint_range:nnn #1
  {
    \exp_args:Nf \__codepoint_range_aux:nnn
      { \int_eval:n { "#1 - \l__codepoint_next_codepoint_fint_tl } }
  }
\cs_set_protected:Npn \__codepoint_range_aux:nnn #1#2
  {
    \exp_args:Nf \__codepoint_range:nnnn
      {
        \int_min:nn
          {#1}
          {
            \c__codepoint_block_size_int
            - \clist_count:c { l__codepoint_ #2 _block_clist }
          }
      }
      {#1} {#2}
  }
```

Here we want to do three things: add to and possibly complete the current block, add complete blocks quickly, then finish up the range in a final open block. We need to avoid as far as possible avoid dealing with every single codepoint, so the middle step is optimised.

```
\cs_set_protected:Npn \__codepoint_range:nnnn #1#2#3#4
  {
    \prg_replicate:nn {#1}
      { \clist_put_right:cn { l__codepoint_ #3 _block_clist } {#4} }
  \int_compare:nNnT { \clist_count:c { l__codepoint_ #3 _block_clist } }
    = \c__codepoint_block_size_int
    { \__codepoint_save_blocks:nn {#3} { 1 } }
  \int_compare:nNnF
    { \int_div_truncate:nn { #2 - #1 } \c__codepoint_block_size_int } = 0
    {
      \tl_set:ce { l__codepoint_ #3 _block_clist }
        {
```

```
          \exp_args:NNe \use:nn \use_none:n
            { \prg_replicate:nn { \c__codepoint_block_size_int } { , #4 } }
        }
      \__codepoint_save_blocks:nn {#3}
        { \int_div_truncate:nn { (#2 - #1) } \c__codepoint_block_size_int }
    }
  \prg_replicate:nn
    { \int_mod:nn { #2 - #1 } \c__codepoint_block_size_int }
    { \clist_put_right:ce { l__codepoint_ #3 _block_clist } {#4} }
  }
```

To allow rapid comparison, each completed block is stored locally as a comma list: once all of the blocks have been created, they are converted into an `intarray` in one step. The aim here is to check the current block against those we've already used, and either match to an existing block or save a new block.

```
\cs_set_protected:Npn \__codepoint_save_blocks:nn #1#2
  {
    \tl_set_eq:Nc \l__codepoint_matched_block_tl { l__codepoint_ #1 _block_tl }
    \int_step_inline:nn { \tl_use:c { l__codepoint_ #1 _block_tl } - 1 }
      {
        \tl_if_eq:ccT { l__codepoint_ #1 _block_clist }
          { l__codepoint_ #1 _block_ ##1 _clist }
          { \tl_set:Nn \l__codepoint_matched_block_tl {##1} }
      }
    \int_compare:nNnT
      { \tl_use:c { l__codepoint_ #1 _block_tl } } = \l__codepoint_matched_block_tl
        {
          \clist_set_eq:cc
            {
              l__codepoint_ #1 _block_
              \tl_use:c { l__codepoint_ #1 _block_tl } _clist
            }
            { l__codepoint_ #1 _block_clist }
          \tl_set:ce { l__codepoint_ #1 _block_tl }
            { \int_eval:n { \tl_use:c { l__codepoint_ #1 _block_tl } + 1 } }
        }
```

Here, we avoid `\prg_replicate:nn` as the number of tokens generated would be high: that shows in the format dump (although TEX recovers memory during the subsequent runs).

```
      \int_step_inline:nnn
        { \tl_use:c { l__codepoint_ #1 _pos_tl } + 1 }
        { \tl_use:c { l__codepoint_ #1 _pos_tl } + #2 }
        {
          \exp_args:Nc \__kernel_intarray_gset:Nnn
            { g__codepoint_ #1 _index_intarray }
            {##1}
            \l__codepoint_matched_block_tl
        }
      \tl_set:ce { l__codepoint_ #1 _pos_tl }
        { \int_eval:n { \tl_use:c { l__codepoint_ #1 _pos_tl } + #2 } }
    \clist_clear:c { l__codepoint_ #1 _block_clist }
  }
```

Close out the final block, rename the first stage table, then combine all of the block comma-lists into one large second-stage table with offsets. As we use an index not an offset, there is a little back-and-forward to do.

```
\cs_set_protected:Npn \__codepoint_finalise_blocks:
  {
    \clist_map_inline:nn { category , uppercase , lowercase }
      {
        \__codepoint_range:nnn { 110000 } {##1} { 0 }
        \__codepoint_finalise_blocks:n {##1}
      }
  }
\cs_set_protected:Npn \__codepoint_finalise_blocks:n #1
  {
    \cs_gset_eq:cc { c__codepoint_ #1 _index_intarray } { g__codepoint_ #1 _index_intarra
    \cs_undefine:c { g__codepoint_ #1 _index_intarray }
    \intarray_new:cn { g__codepoint_ #1 _blocks_intarray }
      { ( \tl_use:c { l__codepoint_ #1 _block_tl } - 1 ) * \c__codepoint_block_size_int }
    \int_step_inline:nn { \tl_use:c { l__codepoint_ #1 _block_tl } - 1 }
      {
        \exp_args:Nv \__codepoint_finalise_blocks:nnn
          { l__codepoint_ #1 _block_ ##1 _clist }
          {##1} {#1}
      }
    \cs_gset_eq:cc { c__codepoint_ #1 _blocks_intarray }
      { g__codepoint_ #1 _blocks_intarray }
    \cs_undefine:c { g__codepoint_ #1 _blocks_intarray }
  }
\cs_set_protected:Npn \__codepoint_finalise_blocks:nnn #1#2#3
  {
    \exp_args:Nnf \__codepoint_finalise_blocks:nnnw { 1 }
      { \int_eval:n { ( #2 - 1 ) * \c__codepoint_block_size_int } }
      {#3}
      #1 , \q_recursion_tail , \q_recursion_stop
  }
\cs_set_protected:Npn \__codepoint_finalise_blocks:nnnw #1#2#3#4 ,
  {
    \quark_if_recursion_tail_stop:n {#4}
    \intarray_gset:cnn { g__codepoint_ #3 _blocks_intarray }
      { #1 + #2 }
      {#4}
    \exp_args:Nf \__codepoint_finalise_blocks:nnnw
      { \int_eval:n { #1 + 1 } } {#2} {#3}
  }
```

With the setup done, read the main data file: it's easiest to do that as a token list with spaces retained.

```
\ior_open:Nn \g__codepoint_data_ior { UnicodeData.txt }
\group_begin:
  \char_set_catcode_space:n { `\  }%
  \ior_map_variable:NNn \g__codepoint_data_ior \l__codepoint_tmpa_tl
    {%
      \if_meaning:w \l__codepoint_tmpa_tl \c_space_tl
        \exp_after:wN \ior_map_break:
      \fi:
```

```
      \exp_after:wN \__codepoint_data_auxi:w \l__codepoint_tmpa_tl \q_stop
    }%
  \__codepoint_finalise_blocks:
 \group_end:
\group_end:
```

__kernel_codepoint_data:nn
__codepoint_data:nnn

Recover data from a two-stage table: entirely generic as this applies to all tables (as we use the same block size for all of them). Notice that as we use indices not offsets we have to shuffle out-by-one issues. This function is needed *before* loading the special casing data, as there we need to be able to check the standard case mappings.

```
\cs_new:Npn \__kernel_codepoint_data:nn #1#2
  {
    \exp_args:Nf \__codepoint_data:nnn
      {
        \int_eval:n
          {
            \c__codepoint_block_size_int *
              (
                \intarray_item:cn { c__codepoint_ #1 _index_intarray }
                  {
                    \int_div_truncate:nn {#2}
                      \c__codepoint_block_size_int
                    + 1
                  }
                  - 1
              )
          }
      }
      {#2} {#1}
  }
\cs_new:Npn \__codepoint_data:nnn #1#2#3
  {
    \intarray_item:cn { c__codepoint_ #3 _blocks_intarray }
      { #1 + \int_mod:nn {#2} \c__codepoint_block_size_int + 1 }
  }
```

(*End of definition for* __kernel_codepoint_data:nn *and* __codepoint_data:nnn*.*)

The other data files all use C-style comments so we have to worry about # tokens (and reading as strings). The set up for case folding is in two parts. For the basic (core) mappings, folding is the same as lower casing in most positions so only store the differences. For the more complex foldings, always store the result, splitting up the two or three code points in the input as required.

```
\group_begin:
  \ior_open:Nn \g__codepoint_data_ior { CaseFolding.txt }
  \cs_set_protected:Npn \__codepoint_data_auxi:w #1 ;~ #2 ;~ #3 ; #4 \q_stop
    {
      \if:w \tl_head:n { #2 ? } C
        \reverse_if:N \if_int_compare:w
          \int_eval:n { \__kernel_codepoint_data:nn { lowercase } {"#1} + "#1 }
            = "#3 ~
          \tl_const:ce
            { c__codepoint_casefold_ \codepoint_str_generate:n {"#1} _tl }
            { {"#3} { } { } }
```

```
      \fi:
    \else:
      \if:w \tl_head:n { #2 ? } F
        \__codepoint_data_auxii:w #1 ~ #3 ~ \q_stop
      \fi:
    \fi:
  }
```

Here, #4 can have a trailing space, so we tidy up a bit at the cost of speed for these small number of cases it applies to.

```
\cs_set_protected:Npn \__codepoint_data_auxii:w #1 ~ #2 ~ #3 ~ #4 \q_stop
  {
    \tl_const:ce { c__codepoint_casefold_ \codepoint_str_generate:n {"#1} _tl }
      {
        {"#2}
        {"#3}
        { \tl_if_blank:nF {#4} { " \int_to_Hex:n {"#4} } }
      }
  }
\ior_str_map_inline:Nn \g__codepoint_data_ior
  {
    \reverse_if:N \if:w \c_hash_str \tl_head:w #1 \c_hash_str \q_stop
      \__codepoint_data_auxi:w #1 \q_stop
    \fi:
  }
\ior_close:N \g__codepoint_data_ior
```

For upper- and lowercasing special situations, there is a bit more to do as we also have titlecasing to consider, plus we need to stop part-way through the file.

```
\ior_open:Nn \g__codepoint_data_ior { SpecialCasing.txt }
\cs_set_protected:Npn \__codepoint_data_auxi:w
  #1 ;~ #2 ;~ #3 ;~ #4 ; #5 \q_stop
  {
    \use:n { \__codepoint_data_auxii:w #1 ~ lower ~ #2 ~ } ~ \q_stop
    \use:n { \__codepoint_data_auxii:w #1 ~ upper ~ #4 ~ } ~ \q_stop
    \str_if_eq:nnF {#3} {#4}
      { \use:n { \__codepoint_data_auxii:w #1 ~ title ~ #3 ~ } ~ \q_stop }
  }
\cs_set_protected:Npn \__codepoint_data_auxii:w
  #1 ~ #2 ~ #3 ~ #4 ~ #5 \q_stop
  {
    \tl_if_empty:nF {#4}
      {
        \tl_const:ce { c__codepoint_ #2 case_ \codepoint_str_generate:n {"#1} _tl }
          {
            {"#3}
            {"#4}
            { \tl_if_blank:nF {#5} {"#5} }
          }
      }
  }
\ior_str_map_inline:Nn \g__codepoint_data_ior
  {
    \str_if_eq:eeTF { \tl_head:w #1 \c_hash_str \q_stop } { \c_hash_str }
      {
```

```
          \str_if_eq:eeT
            {#1}
            { \c_hash_str \c_space_tl Conditional~Mappings }
            { \ior_map_break: }
        }
        { \__codepoint_data_auxi:w #1 \q_stop }
    }
  \ior_close:N \g__codepoint_data_ior
\group_end:
```

__kernel_codepoint_case:nn
__codepoint_case:nnn
__codepoint_uppercase:n
__codepoint_lowercase:n
__codepoint_titlecase:n
__codepoint_casefold:n
__codepoint_case:nn

With the core data files loaded, there is now a need to provide access to this information for other modules. That is done here such that case folding can also be covered. At this level, all that needs to be returned is the

```
\cs_new:Npn \__kernel_codepoint_case:nn #1#2
  {
    \exp_args:Ne \__codepoint_case:nnn
      { \codepoint_str_generate:n {#2} } {#1} {#2}
  }
\cs_new:Npn \__codepoint_case:nnn #1#2#3
  {
    \cs_if_exist:cTF { c__codepoint_ #2 _ #1 _tl }
      {
        \tl_use:c
          { c__codepoint_ #2 _ #1 _tl }
      }
      { \use:c { __codepoint_ #2 :n } {#3} }
  }
\cs_new:Npn \__codepoint_uppercase:n { \__codepoint_case:nn { uppercase } }
\cs_new:Npn \__codepoint_lowercase:n { \__codepoint_case:nn { lowercase } }
\cs_new:Npn \__codepoint_titlecase:n { \__codepoint_case:nn { uppercase } }
\cs_new:Npn \__codepoint_casefold:n  { \__codepoint_case:nn { lowercase } }
\cs_new:Npn \__codepoint_case:nn #1#2
  {
    { \int_eval:n { \__kernel_codepoint_data:nn {#1} {#2} + #2 } }
    { }
    { }
  }
```

(*End of definition for* `\__kernel_codepoint_case:nn` *and others.*)

__codepoint_nfd:n
__codepoint_nfd:nn

A simple interface.

```
\cs_new:Npn \__codepoint_nfd:n #1
  { \exp_args:Ne \__codepoint_nfd:nn { \codepoint_str_generate:n {#1} } {#1} }
\cs_new:Npn \__codepoint_nfd:nn #1#2
  {
    \tl_if_exist:cTF { c__codepoint_nfd_ #1 _tl }
      { \tl_use:c { c__codepoint_nfd_ #1 _tl } }
      { {#2} { } }
  }
```

(*End of definition for* `\__codepoint_nfd:n` *and* `\__codepoint_nfd:nn`.)

```
⟨@@=text⟩
```

Read the Unicode grapheme data. This is quite easy to handle and we only need codepoints, not characters, so there is no need to worry about the engine in use. As reading as a string is most convenient, we have to do some work to remove spaces: the hardest part of the entire process!

```
\ior_new:N \g__text_data_ior
\group_begin:
  \ior_open:Nn \g__text_data_ior { GraphemeBreakProperty.txt }
  \cs_set_nopar:Npn \l__text_tmpa_str { }
  \cs_set_nopar:Npn \l__text_tmpb_str { }
  \cs_set_protected:Npn \__text_data_auxi:w #1 ;~ #2 ~ #3 \q_stop
    {
      \str_if_eq:VnF \l__text_tmpb_str {#2}
        {
          \str_if_empty:NF \l__text_tmpb_str
            {
              \clist_const:ce { c__text_grapheme_ \l__text_tmpb_str _clist }
                { \exp_after:wN \use_none:n \l__text_tmpa_str }
              \cs_set_nopar:Npn \l__text_tmpa_str { }
            }
          \cs_set_nopar:Npn \l__text_tmpb_str {#2}
        }
      \__text_data_auxii:w #1 .. #1 .. #1 \q_stop
    }
  \cs_set_protected:Npn \__text_data_auxii:w #1 .. #2 .. #3 \q_stop
    {
      \cs_set_nopar:Npe \l__text_tmpa_str
        {
          \l__text_tmpa_str ,
          \tl_trim_spaces:n {#1} .. \tl_trim_spaces:n {#2}
        }
    }
  \ior_str_map_inline:Nn \g__text_data_ior
    {
      \str_if_eq:eeF { \tl_head:w #1 \c_hash_str \q_stop } { \c_hash_str }
        {
          \tl_if_blank:nF {#1}
            { \__text_data_auxi:w #1 \q_stop }
        }
    }
  \ior_close:N \g__text_data_ior
\group_end:
⟨/package⟩
```

Chapter 87

l3text implementation

```
⟨*package⟩
⟨@@=text⟩
\cs_generate_variant:Nn \tl_if_head_eq_meaning_p:nN { o }
```

87.1 Internal auxiliaries

\s__text_stop

Internal scan marks.

```
\scan_new:N \s__text_stop
```

(*End of definition for* `\s__text_stop`.)

\q__text_nil

Internal quarks.

```
\quark_new:N \q__text_nil
```

(*End of definition for* `\q__text_nil`.)

__text_quark_if_nil_p:n
__text_quark_if_nil:nTF

Branching quark conditional.

```
\__kernel_quark_new_conditional:Nn \__text_quark_if_nil:n { TF }
```

(*End of definition for* `\__text_quark_if_nil:nTF`.)

\q__text_recursion_tail
\q__text_recursion_stop

Internal recursion quarks.

```
\quark_new:N \q__text_recursion_tail
\quark_new:N \q__text_recursion_stop
```

(*End of definition for* `\q__text_recursion_tail` *and* `\q__text_recursion_stop`.)

__text_use_i_delimit_by_q_recursion_stop:nw

Functions to gobble up to a quark.

```
\cs_new:Npn \__text_use_i_delimit_by_q_recursion_stop:nw
  #1 #2 \q__text_recursion_stop {#1}
```

(*End of definition for* `\__text_use_i_delimit_by_q_recursion_stop:nw`.)

__text_if_q_recursion_tail_stop_do:Nn
__text_if_q_recursion_tail_stop_do:nn

Functions to query recursion quarks.

```
\__kernel_quark_new_test:N \__text_if_q_recursion_tail_stop_do:Nn
\__kernel_quark_new_test:N \__text_if_q_recursion_tail_stop_do:nn
```

(*End of definition for* `\__text_if_q_recursion_tail_stop_do:Nn` *and* `\__text_if_q_recursion_tail_-stop_do:nn`.)

\s__text_recursion_tail
\s__text_recursion_stop

Internal scan marks quarks.

```
\scan_new:N \s__text_recursion_tail
\scan_new:N \s__text_recursion_stop
```

(*End of definition for* `\s__text_recursion_tail` *and* `\s__text_recursion_stop`.)

__text_use_i_delimit_by_s_recursion_stop:nw

Functions to gobble up to a scan mark.

```
\cs_new:Npn \__text_use_i_delimit_by_s_recursion_stop:nw
  #1 #2 \s__text_recursion_stop {#1}
```

(*End of definition for* `\__text_use_i_delimit_by_s_recursion_stop:nw`.)

__text_if_s_recursion_tail_stop_do:Nn

Functions to query recursion scan marks. Slower than a quark test but needed to avoid issues in the outer expansion loop with unterminated `\romannumeral` primitives.

```
\cs_new:Npn \__text_if_s_recursion_tail_stop_do:Nn #1
  {
    \bool_lazy_and:nnTF
      { \cs_if_eq_p:NN \s__text_recursion_tail #1 }
      { \str_if_eq_p:nn { \s__text_recursion_tail } {#1} }
      { \__text_use_i_delimit_by_s_recursion_stop:nw }
      { \use_none:n }
  }
```

(*End of definition for* `\__text_if_s_recursion_tail_stop_do:Nn`.)

87.2 Utilities

__text_token_to_explicit:N
__text_token_to_explicit_char:N
__text_token_to_explicit_cs:N
__text_token_to_explicit_cs_aux:N
__text_token_to_explicit:n
__text_token_to_explicit_auxi:w
__text_token_to_explicit_auxii:w
__text_token_to_explicit_auxiii:w

The idea here is to take a token and ensure that if it's an implicit char, we output the explicit version. Otherwise, the token needs to be unchanged. First, we have to split between control sequences and everything else.

```
\group_begin:
  \char_set_catcode_active:n { 0 }
  \cs_new:Npn \__text_token_to_explicit:N #1
    {
      \if_catcode:w \exp_not:N #1
        \if_catcode:w \scan_stop: \exp_not:N #1
          \scan_stop:
        \else:
          \exp_not:N ^^@
        \fi:
        \exp_after:wN \__text_token_to_explicit_cs:N
      \else:
        \exp_after:wN \__text_token_to_explicit_char:N
      \fi:
      #1
    }
\group_end:
```

For control sequences, we can check for macros versus other cases using `\if_meaning:w`, then explicitly check for `\chardef` and `\mathchardef`.

```
\cs_new:Npn \__text_token_to_explicit_cs:N #1
  {
    \exp_after:wN \if_meaning:w \exp_not:N #1 #1
      \exp_after:wN \use:nn \exp_after:wN
        \__text_token_to_explicit_cs_aux:N
    \else:
      \exp_after:wN \exp_not:n
    \fi:
      {#1}
  }
\cs_new:Npn \__text_token_to_explicit_cs_aux:N #1
  {
    \bool_lazy_or:nnTF
      { \token_if_chardef_p:N #1 }
      { \token_if_mathchardef_p:N #1 }
      {
        \char_generate:nn {#1}
          {
            \if_int_compare:w \char_value_catcode:n {#1} = 10 \exp_stop_f:
              10
            \else:
              12
            \fi:
          }
      }
      {#1}
  }
```

For character tokens, we need to filter out the implicit characters from those that are explicit. That's done here, then if necessary we work out the category code and generate the char. To avoid issues with alignment tabs, that one is done by elimination rather than looking up the code explicitly. The trick with finding the charcode is that the TEX messages are either `the` ⟨*something*⟩ `character` ⟨*char*⟩ or `the` ⟨*type*⟩ ⟨*char*⟩.

```
\cs_new:Npn \__text_token_to_explicit_char:N #1
  {
    \if:w
      \if_catcode:w ^ \exp_args:No \str_tail:n { \token_to_str:N #1 } ^
        \token_to_str:N #1 #1
        \else:
        AB
      \fi:
      \exp_after:wN \exp_not:n
    \else:
      \exp_after:wN \__text_token_to_explicit:n
    \fi:
      {#1}
  }
\cs_new:Npn \__text_token_to_explicit:n #1
  {
    \exp_after:wN \__text_token_to_explicit_auxi:w
      \int_value:w
        \if_catcode:w \c_group_begin_token #1 1 \else:
```

```
        \if_catcode:w \c_group_end_token #1 2 \else:
        \if_catcode:w \c_math_toggle_token #1 3 \else:
        \if_catcode:w ## #1 6 \else:
        \if_catcode:w ^ #1 7 \else:
        \if_catcode:w \c_math_subscript_token #1 8 \else:
        \if_catcode:w \c_space_token #1 10 \else:
        \if_catcode:w A #1 11 \else:
        \if_catcode:w + #1 12 \else:
        4 \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi:
    \exp_after:wN ;
    \token_to_meaning:N #1 \s__text_stop
  }
\cs_new:Npn \__text_token_to_explicit_auxi:w #1 ; #2 \s__text_stop
  {
    \char_generate:nn
      {
        \if_int_compare:w #1 < 9 \exp_stop_f:
          \exp_after:wN \__text_token_to_explicit_auxii:w
        \else:
          \exp_after:wN \__text_token_to_explicit_auxiii:w
        \fi:
        #2
      }
      {#1}
  }
\exp_last_unbraced:NNNNo \cs_new:Npn \__text_token_to_explicit_auxii:w
  #1 { \tl_to_str:n { character ~ } } { ' }
\cs_new:Npn \__text_token_to_explicit_auxiii:w #1 ~ #2 ~ { ' }
```

(*End of definition for* `\__text_token_to_explicit:N` *and others.*)

`\__text_char_catcode:N` An idea from l3char: we need to get the category code of a specific token, not the general case.

```
\cs_new:Npn \__text_char_catcode:N #1
  {
    \if_catcode:w \exp_not:N #1 \c_math_toggle_token
      3
    \else:
      \if_catcode:w \exp_not:N #1 \c_alignment_token
        4
      \else:
        \if_catcode:w \exp_not:N #1 \c_math_superscript_token
          7
        \else:
          \if_catcode:w \exp_not:N #1 \c_math_subscript_token
            8
          \else:
            \if_catcode:w \exp_not:N #1 \c_space_token
              10
            \else:
              \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
                11
              \else:
                \if_catcode:w \exp_not:N #1 \c_catcode_other_token
```

```
                  12
                \else:
                  13
                \fi:
              \fi:
            \fi:
          \fi:
        \fi:
      \fi:
    \fi:
  }
```

(*End of definition for* `\__text_char_catcode:N`.)

`\__text_if_expandable:NTF`

Test for tokens that make sense to expand here: that is more restrictive than the engine view.

```
\prg_new_conditional:Npnn \__text_if_expandable:N #1 { T , F , TF }
  {
    \token_if_expandable:NTF #1
      {
        \bool_lazy_any:nTF
          {
            { \token_if_protected_macro_p:N      #1 }
            { \token_if_protected_long_macro_p:N #1 }
            { \token_if_eq_meaning_p:NN \q__text_recursion_tail #1 }
          }
          { \prg_return_false: }
          { \prg_return_true: }
      }
      { \prg_return_false: }
  }
```

(*End of definition for* `\__text_if_expandable:NTF`.)

87.3 Codepoint utilities

For working with codepoints in an engine-neutral way.

`\__text_codepoint_process:nN`
`\__text_codepoint_process_aux:nN`
`\__text_codepoint_process:nNN`
`\__text_codepoint_process:nNNN`
`\__text_codepoint_process:nNNNN`

Grab a codepoint and apply some code to it: here `#1` should expect one following *balanced text*.

```
\bool_lazy_or:nnTF
  { \sys_if_engine_luatex_p: }
  { \sys_if_engine_xetex_p: }
  {
    \cs_new:Npn \__text_codepoint_process:nN #1#2 { #1 {#2} }
  }
  {
    \cs_new:Npe \__text_codepoint_process:nN #1#2
      {
        \exp_not:N \int_compare:nNnTF {`#2} > { "80 }
          {
            \sys_if_engine_pdftex:TF
              { \exp_not:N \__text_codepoint_process_aux:nN }
```

```
              {
                \exp_not:N \int_compare:nNnTF {‘#2} > { "FF }
                  { \exp_not:N \use:n }
                  { \exp_not:N \__text_codepoint_process_aux:nN }
              }
          }
          { \exp_not:N \use:n }
            {#1} #2
      }
    \cs_new:Npn \__text_codepoint_process_aux:nN #1#2
      {
        \int_compare:nNnTF { ‘#2 } < { "E0 }
          { \__text_codepoint_process:nNN }
          {
            \int_compare:nNnTF { ‘#2 } < { "F0 }
              { \__text_codepoint_process:nNNN }
              { \__text_codepoint_process:nNNNN }
          }
            {#1} #2
        }
    \cs_new:Npn \__text_codepoint_process:nNN #1#2#3
      { #1 {#2#3} }
    \cs_new:Npn \__text_codepoint_process:nNNN #1#2#3#4
      { #1 {#2#3#4} }
    \cs_new:Npn \__text_codepoint_process:nNNNN #1#2#3#4#5
      { #1 {#2#3#4#5} }
  }
```

(*End of definition for* `\__text_codepoint_process:nN` *and others.*)

__text_codepoint_compare_p:nNn
__text_codepoint_compare:nNn*TF*
__text_codepoint_from_chars:Nw
__text_codepoint_from_chars_aux:Nw
__text_codepoint_from_chars:N
__text_codepoint_from_chars:NN
__text_codepoint_from_chars:NNN
__text_codepoint_from_chars:NNNN

Allows comparison for all engines using a first "character" followed by a codepoint.

```
\bool_lazy_or:nnTF
  { \sys_if_engine_luatex_p: }
  { \sys_if_engine_xetex_p: }
  {
    \prg_new_conditional:Npnn
      \__text_codepoint_compare:nNn #1#2#3 { TF , p }
      {
        \int_compare:nNnTF {‘#1} #2 {#3}
          \prg_return_true: \prg_return_false:
      }
    \cs_new:Npn \__text_codepoint_from_chars:Nw #1 {‘#1}
  }
  {
    \prg_new_conditional:Npnn
      \__text_codepoint_compare:nNn #1#2#3 { TF , p }
      {
        \int_compare:nNnTF { \__text_codepoint_from_chars:Nw #1 }
            #2 {#3}
          \prg_return_true: \prg_return_false:
      }
    \cs_new:Npe \__text_codepoint_from_chars:Nw #1
      {
        \exp_not:N \if_int_compare:w ‘#1 > "80 \exp_not:N \exp_stop_f:
```

```
    \sys_if_engine_pdftex:TF
     {
      \exp_not:N \exp_after:wN
       \exp_not:N \__text_codepoint_from_chars_aux:Nw
     }
     {
      \exp_not:N \if_int_compare:w `#1 > "FF \exp_not:N \exp_stop_f:
       \exp_not:N \exp_after:wN \exp_not:N \exp_after:wN
        \exp_not:N \exp_after:wN
        \exp_not:N \__text_codepoint_from_chars:N
      \exp_not:N \else:
       \exp_not:N \exp_after:wN \exp_not:N \exp_after:wN
        \exp_not:N \exp_after:wN
        \exp_not:N \__text_codepoint_from_chars_aux:Nw
      \exp_not:N \fi:
     }
   \exp_not:N \else:
    \exp_not:N \exp_after:wN \exp_not:N \__text_codepoint_from_chars:N
   \exp_not:N \fi:
    #1
  }
 \cs_new:Npn \__text_codepoint_from_chars_aux:Nw #1
  {
   \if_int_compare:w `#1 < "E0 \exp_stop_f:
    \exp_after:wN \__text_codepoint_from_chars:NN
   \else:
    \if_int_compare:w `#1 < "F0 \exp_stop_f:
     \exp_after:wN \exp_after:wN \exp_after:wN
      \__text_codepoint_from_chars:NNN
    \else:
     \exp_after:wN \exp_after:wN \exp_after:wN
      \__text_codepoint_from_chars:NNNN
    \fi:
   \fi:
    #1
  }
 \cs_new:Npn \__text_codepoint_from_chars:N #1 {`#1}
 \cs_new:Npn \__text_codepoint_from_chars:NN #1#2
  { (`#1 - "C0) * "40 + `#2 - "80 }
 \cs_new:Npn \__text_codepoint_from_chars:NNN #1#2#3
  { (`#1 - "E0) * "1000 + (`#2 - "80) * "40 + `#3 - "80 }
 \cs_new:Npn \__text_codepoint_from_chars:NNNN #1#2#3#4
  {
    (`#1 - "F0) * "40000
   + (`#2 - "80) * "1000
   + (`#3 - "80) * "40
   + `#4 - "80
  }
}
```

(*End of definition for* `\__text_codepoint_compare:nNnTF` *and others.*)

87.4 Configuration variables

\l_text_accents_tl
\l_text_letterlike_tl

Used to be used for excluding these ideas from expansion: now deprecated.

```
\tl_new:N \l_text_accents_tl
\tl_new:N \l_text_letterlike_tl
```

(*End of definition for* `\l_text_accents_tl` *and* `\l_text_letterlike_tl`*.*)

\l_text_case_exclude_arg_tl

Non-text arguments, including covering the case of `\protected@edef` applied to `\cite`.

```
\tl_new:N \l_text_case_exclude_arg_tl
\tl_set:Ne \l_text_case_exclude_arg_tl
  {
    \exp_not:n { \begin \cite \end \label \ref }
    \exp_not:c { cite ~ }
    \exp_not:n { \babelshorthand }
  }
```

(*End of definition for* `\l_text_case_exclude_arg_tl`*. This variable is documented on page 298.*)

\l_text_math_arg_tl

Math mode as arguments.

```
\tl_new:N \l_text_math_arg_tl
\tl_set:Nn \l_text_math_arg_tl { \ensuremath }
```

(*End of definition for* `\l_text_math_arg_tl`*. This variable is documented on page 298.*)

\l_text_math_delims_tl

Paired math mode delimiters.

```
\tl_new:N \l_text_math_delims_tl
\tl_set:Nn \l_text_math_delims_tl { $ $ \( \) }
```

(*End of definition for* `\l_text_math_delims_tl`*. This variable is documented on page 298.*)

\l_text_expand_exclude_tl

Commands which need not to expand. We start with a somewhat historical list, and tidy up if possible.

```
\tl_new:N \l_text_expand_exclude_tl
\tl_set:Nn \l_text_expand_exclude_tl
  { \begin \cite \end \label \ref }
\bool_lazy_and:nnT
  { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
  { \tl_if_exist_p:N \@expl@finalise@setup@@ }
  {
    \tl_gput_right:Nn \@expl@finalise@setup@@
      {
        \tl_gput_right:Nn \@kernel@after@begindocument
          {
            \group_begin:
              \cs_set_protected:Npn \__text_tmp:w #1
                {
                  \tl_clear:N \l_text_expand_exclude_tl
                  \tl_map_inline:nn {#1}
                    {
                      \bool_lazy_any:nF
                        {
                          { \token_if_protected_macro_p:N ##1 }
                          { \token_if_protected_long_macro_p:N ##1 }
                          {
```

```
                            \str_if_eq_p:ee
                              { \cs_replacement_spec:N ##1 }
                              { \exp_not:n { \protect ##1 } \c_space_tl }
                          }
                        }
                        { \tl_put_right:Nn \l_text_expand_exclude_tl {##1} }
                    }
                }
              \exp_args:NV \__text_tmp:w \l_text_expand_exclude_tl
            \exp_args:NNNV \group_end:
            \tl_set:Nn \l_text_expand_exclude_tl \l_text_expand_exclude_tl
          }
      }
  }
```

(*End of definition for* `\l_text_expand_exclude_tl`. *This variable is documented on page* *298*.)

`\l__text_math_mode_tl`

Used to control math mode output: internal as there is a dedicated setter.

```
\tl_new:N \l__text_math_mode_tl
```

(*End of definition for* `\l__text_math_mode_tl`.)

87.5 Expansion to formatted text

`\c__text_chardef_space_token`
`\c__text_mathchardef_space_token`
`\c__text_chardef_group_begin_token`
`\c__text_mathchardef_group_begin_token`
`\c__text_chardef_group_end_token`
`\c__text_mathchardef_group_end_token`

Markers for implicit char handling.

```
\tex_global:D \tex_chardef:D \c__text_chardef_space_token = `\  %
\tex_global:D \tex_mathchardef:D \c__text_mathchardef_space_token = `\  %
\tex_global:D \tex_chardef:D \c__text_chardef_group_begin_token = `\{ % `\}
\tex_global:D \tex_mathchardef:D \c__text_mathchardef_group_begin_token = `\{ % `\} `\{
\tex_global:D \tex_chardef:D \c__text_chardef_group_end_token = `\} % `\{
\tex_global:D \tex_mathchardef:D \c__text_mathchardef_group_end_token = `\} %
```

(*End of definition for* `\c__text_chardef_space_token` *and others.*)

`\text_expand:n`
`\__text_expand:n`
`\__text_expand_result:n`
`\__text_expand_store:n`
`\__text_expand_store:o`
`\__text_expand_store:nw`
`\__text_expand_end:w`
`\__text_expand_loop:w`
`\__text_expand_group:n`
`\__text_expand_space:w`
`\__text_expand_N_type:N`
`\__text_expand_math_search:NNN`
`\__text_expand_math_loop:Nw`
`\__text_expand_math_N_type:NN`
`\__text_expand_math_group:Nn`
`\__text_expand_math_space:Nw`
`\__text_expand_explicit:N`
`\__text_expand_exclude:N`
`\__text_expand_exclude_switch:Nnnnn`
`\__text_expand_exclude:nN`
`\__text_expand_exclude:NN`
`\__text_expand_exclude:Nw`
`\__text_expand_exclude:Nnn`
`\__text_expand_accent:N`
`\__text_expand_accent:NN`
`\__text_expand_letterlike:N`
`\__text_expand_letterlike:NN`
`\__text_expand_cs:N`

After precautions against & tokens, start a simple loop: that of course means that "text" cannot contain the two recursion quarks. The loop here must be `f`-type expandable; we have arbitrary user commands which might be protected *and* take arguments, and if the expansion code is used in a typesetting context, that will otherwise explode. (The same issue applies more clearly to case changing: see the example there.) The outer loop has to use scan marks as delimiters to protect against unterminated `\romannumeral` usage in the input.

```
\cs_new:Npn \text_expand:n #1
  {
    \__kernel_exp_not:w \exp_after:wN
      {
        \exp:w
        \__text_expand:n {#1}
      }
  }
\cs_new:Npn \__text_expand:n #1
  {
    \group_align_safe_begin:
    \__text_expand_loop:w #1
```

```
      \s__text_recursion_tail \s__text_recursion_stop
    \__text_expand_result:n { }
  }
```

The approach to making the code f-type expandable is to usee a marker result token and to shuffle the collected tokens

```
\cs_new:Npn \__text_expand_store:n #1
  { \__text_expand_store:nw {#1} }
\cs_generate_variant:Nn \__text_expand_store:n { o }
\cs_new:Npn \__text_expand_store:nw #1#2 \__text_expand_result:n #3
  { #2 \__text_expand_result:n { #3 #1 } }
\cs_new:Npn \__text_expand_end:w #1 \__text_expand_result:n #2
  {
    \group_align_safe_end:
    \exp_end:
    #2
  }
```

The main loop is a standard "tl action"; groups are handled recursively, while spaces are just passed through. Thus all of the action is in handling N-type tokens.

```
\cs_new:Npn \__text_expand_loop:w #1 \s__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#1}
      { \__text_expand_N_type:N }
      {
        \tl_if_head_is_group:nTF {#1}
          { \__text_expand_group:n }
          { \__text_expand_space:w }
      }
    #1 \s__text_recursion_stop
  }
\cs_new:Npn \__text_expand_group:n #1
  {
    \__text_expand_store:o
      {
        \exp_after:wN
          {
            \exp:w
            \__text_expand:n {#1}
          }
      }
    \__text_expand_loop:w
  }
\exp_last_unbraced:NNo \cs_new:Npn \__text_expand_space:w \c_space_tl
  {
    \__text_expand_store:n { ~ }
    \__text_expand_loop:w
  }
```

The first step in dealing with N-type tokens is to look for math mode material: that needs to be left alone. The starting function has to be split into two as we need `\quark_if_recursion_tail_stop:N` first before we can trigger the search. We then look for matching pairs of delimiters, allowing for the case where math mode starts but does not end. Within math mode, we simply pass all the tokens through unchanged, just checking the N-type ones against the end marker.

```
\cs_new:Npn \__text_expand_N_type:N #1
  {
    \__text_if_s_recursion_tail_stop_do:Nn #1
      { \__text_expand_end:w }
    \exp_after:wN \__text_expand_math_search:NNN
      \exp_after:wN #1 \l_text_math_delims_tl
      \q__text_recursion_tail \q__text_recursion_tail
      \q__text_recursion_stop
  }
\cs_new:Npn \__text_expand_math_search:NNN #1#2#3
  {
    \__text_if_q_recursion_tail_stop_do:Nn #2
      { \__text_expand_explicit:N #1 }
    \token_if_eq_meaning:NNTF #1 #2
      {
        \__text_use_i_delimit_by_q_recursion_stop:nw
          {
            \__text_expand_store:n {#1}
            \__text_expand_math_loop:Nw #3
          }
      }
      { \__text_expand_math_search:NNN #1 }
  }
\cs_new:Npn \__text_expand_math_loop:Nw #1#2 \s__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#2}
      { \__text_expand_math_N_type:NN }
      {
        \tl_if_head_is_group:nTF {#2}
          { \__text_expand_math_group:Nn }
          { \__text_expand_math_space:Nw }
      }
    #1#2 \s__text_recursion_stop
  }
\cs_new:Npn \__text_expand_math_N_type:NN #1#2
  {
    \__text_if_s_recursion_tail_stop_do:Nn #2
      { \__text_expand_end:w }
    \token_if_eq_meaning:NNF #2 \exp_not:N
      { \__text_expand_store:n {#2} }
    \token_if_eq_meaning:NNTF #2 #1
      { \__text_expand_loop:w }
      { \__text_expand_math_loop:Nw #1 }
  }
\cs_new:Npn \__text_expand_math_group:Nn #1#2
  {
    \__text_expand_store:n { {#2} }
    \__text_expand_math_loop:Nw #1
  }
\exp_after:wN \cs_new:Npn \exp_after:wN \__text_expand_math_space:Nw
  \exp_after:wN # \exp_after:wN 1 \c_space_tl
  {
    \__text_expand_store:n { ~ }
    \__text_expand_math_loop:Nw #1
```

```
  }
```

At this stage, either we have a control sequence or a simple character: split and handle. The need to check for non-protected actives arises from handling of legacy input encodings: they need to end up in a representation we can deal with in further processing. The tests for explicit parts of the LaTeX 2_ε UTF-8 mechanism cover the case of bookmarks, where definitions change and are no longer protected. The same is true for babel shorthands.

```
\cs_new:Npn \__text_expand_explicit:N #1
  {
    \token_if_cs:NTF #1
      { \__text_expand_exclude:N #1 }
      {
        \bool_lazy_and:nnTF
          { \token_if_active_p:N #1 }
          {
            ! \bool_lazy_any_p:n
              {
                { \token_if_protected_macro_p:N #1 }
                { \token_if_protected_long_macro_p:N #1 }
                { \tl_if_head_eq_meaning_p:oN {#1} \UTFviii@two@octets }
                { \tl_if_head_eq_meaning_p:oN {#1} \UTFviii@three@octets }
                { \tl_if_head_eq_meaning_p:oN {#1} \UTFviii@four@octets }
                { \tl_if_head_eq_meaning_p:oN {#1} \active@prefix }
              }
          }
          { \exp_after:wN \__text_expand_loop:w #1 }
          {
            \__text_expand_store:n {#1}
            \__text_expand_loop:w
          }
      }
  }
```

Next we exclude math commands: this is mainly as there *might* be an `\ensuremath`. The switching command for case needs special handling as it has to work by meaning.

```
\cs_new:Npn \__text_expand_exclude:N #1
  {
    \cs_if_eq:NNTF #1 \text_case_switch:nnnn
      { \__text_expand_exclude_switch:Nnnnn #1 }
      {
        \exp_args:Ne \__text_expand_exclude:nN
          {
            \exp_not:V \l_text_math_arg_tl
            \exp_not:V \l_text_expand_exclude_tl
            \exp_not:V \l_text_case_exclude_arg_tl
          }
        #1
      }
  }
\cs_new:Npn \__text_expand_exclude_switch:Nnnnn #1#2#3#4#5
  {
    \__text_expand_store:n { #1 {#2} {#3} {#4} {#5} }
    \__text_expand_loop:w
```

```
  }
\cs_new:Npn \__text_expand_exclude:nN #1#2
  {
    \__text_expand_exclude:NN #2 #1
      \q__text_recursion_tail \q__text_recursion_stop
  }
\cs_new:Npn \__text_expand_exclude:NN #1#2
  {
    \__text_if_q_recursion_tail_stop_do:Nn #2
      { \__text_expand_accent:N #1 }
    \str_if_eq:nnTF {#1} {#2}
      {
        \__text_use_i_delimit_by_q_recursion_stop:nw
          { \__text_expand_exclude:Nw #1 }
      }
      { \__text_expand_exclude:NN #1 }
  }
\cs_new:Npn \__text_expand_exclude:Nw #1#2#
  { \__text_expand_exclude:Nnn #1 {#2} }
\cs_new:Npn \__text_expand_exclude:Nnn #1#2#3
  {
    \__text_expand_store:n { #1#2 {#3} }
    \__text_expand_loop:w
  }
```

Accents.

```
\cs_new:Npn \__text_expand_accent:N #1
  {
    \exp_after:wN \__text_expand_accent:NN \exp_after:wN
      #1 \l_text_accents_tl
      \q__text_recursion_tail \q__text_recursion_stop
  }
\cs_new:Npn \__text_expand_accent:NN #1#2
  {
    \__text_if_q_recursion_tail_stop_do:Nn #2
      { \__text_expand_letterlike:N #1 }
    \cs_if_eq:NNTF #2 #1
      {
        \__text_use_i_delimit_by_q_recursion_stop:nw
          {
            \__text_expand_store:n {#1}
            \__text_expand_loop:w
          }
      }
      { \__text_expand_accent:NN #1 }
  }
```

Another list of exceptions: these ones take no arguments so are easier to handle.

```
\cs_new:Npn \__text_expand_letterlike:N #1
  {
    \exp_after:wN \__text_expand_letterlike:NN \exp_after:wN
      #1 \l_text_letterlike_tl
      \q__text_recursion_tail \q__text_recursion_stop
  }
\cs_new:Npn \__text_expand_letterlike:NN #1#2
```

```
  {
    \__text_if_q_recursion_tail_stop_do:Nn #2
      { \__text_expand_cs:N #1 }
    \cs_if_eq:NNTF #2 #1
      {
        \__text_use_i_delimit_by_q_recursion_stop:nw
          {
            \__text_expand_store:n {#1}
            \__text_expand_loop:w
          }
      }
      { \__text_expand_letterlike:NN #1 }
  }
```

LaTeX 2_ε's `\protect` makes life interesting. Where possible, we simply remove it and replace with the "parent" command; of course, the `\protect` might be explicit, in which case we need to leave it alone. That includes the case where it's not even followed by an `N`-type token. There is also the case of a straight `\@protected@testopt` to cover.

```
\cs_new:Npe \__text_expand_cs:N #1
  {
    \exp_not:N \str_if_eq:nnTF {#1} { \exp_not:N \protect }
      { \exp_not:N \__text_expand_protect:w }
      {
        \bool_lazy_and:nnTF
          { \cs_if_exist_p:N \fmtname }
          { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
          { \exp_not:N \__text_expand_testopt:N #1 }
          { \exp_not:N \__text_expand_replace:N #1 }
      }
  }
\cs_new:Npn \__text_expand_protect:w #1 \s__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#1}
      { \__text_expand_protect:N }
      {
        \__text_expand_store:n { \protect }
        \__text_expand_loop:w
      }
        #1 \s__text_recursion_stop
  }
\cs_new:Npn \__text_expand_protect:N #1
  {
    \__text_if_s_recursion_tail_stop_do:Nn #1
      {
        \__text_expand_store:n { \protect }
        \__text_expand_end:w
      }
    \exp_args:Ne \__text_expand_protect:nN
      { \cs_to_str:N #1 } #1
  }
\cs_new:Npn \__text_expand_protect:nN #1#2
  { \__text_expand_protect:Nw #2 #1 \q__text_nil #1 ~ \q__text_nil \q__text_nil \s__text_st
\cs_new:Npn \__text_expand_protect:Nw #1 #2 ~ \q__text_nil #3 \q__text_nil #4 \s__text_stop
  {
```

```
    \__text_quark_if_nil:nTF {#4}
      {
        \cs_if_exist:cTF {#2}
          { \exp_args:Ne \__text_expand_store:n { \exp_not:c {#2} } }
          { \__text_expand_store:n { \protect #1 } }
      }
      { \__text_expand_store:n { \protect #1 } }
    \__text_expand_loop:w
  }
\cs_new:Npn \__text_expand_testopt:N #1
  {
    \token_if_eq_meaning:NNTF #1 \@protected@testopt
      { \__text_expand_testopt:NNn }
      { \__text_expand_encoding:N #1 }
  }
\cs_new:Npn \__text_expand_testopt:NNn #1#2#3
  {
    \__text_expand_store:n {#1}
    \__text_expand_loop:w
  }
```

Deal with encoding-specific commands

```
\cs_new:Npn \__text_expand_encoding:N #1
  {
    \bool_lazy_or:nnTF
      { \cs_if_eq_p:NN  #1 \@current@cmd }
      { \cs_if_eq_p:NN  #1 \@changed@cmd }
      { \exp_after:wN \__text_expand_loop:w \__text_expand_encoding_escape:NN }
      { \__text_expand_replace:N #1 }
  }
\cs_new:Npn \__text_expand_encoding_escape:NN #1#2 { \exp_not:n {#1} }
```

See if there is a dedicated replacement, and if there is, insert it.

```
\cs_new:Npn \__text_expand_replace:N #1
  {
    \bool_lazy_and:nnTF
      { \cs_if_exist_p:c { l__text_expand_ \token_to_str:N #1 _tl } }
      {
        \bool_lazy_or_p:nn
          { \token_if_cs_p:N #1 }
          { \token_if_active_p:N #1 }
      }
      {
        \exp_args:Nv \__text_expand_replace:n
          { l__text_expand_ \token_to_str:N #1 _tl }
      }
      { \__text_expand_cs_expand:N #1 }
  }
\cs_new:Npn \__text_expand_replace:n #1 { \__text_expand_loop:w #1 }
```

Finally, expand any macros which can be: this then loops back around to deal with what they produce. The only issue is if the token is `\exp_not:n`, as that must apply to the following balanced text.

```
\cs_new:Npn \__text_expand_cs_expand:N #1
  {
```

```
    \__text_if_expandable:NTF #1
      {
        \token_if_eq_meaning:NNTF #1 \exp_not:n
          { \__text_expand_unexpanded:w }
          { \exp_after:wN \__text_expand_loop:w #1 }
      }
      {
        \__text_expand_store:n {#1}
        \__text_expand_loop:w
      }
  }
```

Since `\exp_not:n` is actually a primitive, it allows a strange syntax and it particular the primitive expands what follows and discards spaces and `\scan_stop:` until finding a braced argument (the opening brace can be implicit but we will not support this here). Here, we repeatedly `f`-expand after such an `\exp_not:n`, and test what follows. If it is a brace group, then we found the intended argument of `\exp_not:n`. If it is a space, then the next `f`-expansion will eliminate it. If it is an `N`-type token then `\__text_expand_unexpanded:N` leaves the token to be expanded if it is expandable, and otherwise removes it, assuming that it is `\scan_stop:`. This silently hides errors when `\exp_not:n` is incorrectly followed by some non-expandable token other than `\scan_stop:`, but this should be pretty rare, and there is no good error recovery anyways.

```
\cs_new:Npn \__text_expand_unexpanded:w
  {
    \exp_after:wN \__text_expand_unexpanded_test:w
    \exp:w \exp_end_continue_f:w
  }
\cs_new:Npn \__text_expand_unexpanded_test:w #1 \s__text_recursion_stop
  {
    \tl_if_head_is_group:nTF {#1}
      { \__text_expand_unexpanded:n }
      {
        \__text_expand_unexpanded:w
        \tl_if_head_is_N_type:nT {#1} { \__text_expand_unexpanded:N }
      }
    #1 \s__text_recursion_stop
  }
\cs_new:Npn \__text_expand_unexpanded:N #1
  {
    \exp_after:wN \if_meaning:w \exp_not:N #1 #1
    \else:
      \exp_after:wN #1
    \fi:
  }
\cs_new:Npn \__text_expand_unexpanded:n #1
  {
    \__text_expand_store:n {#1}
    \__text_expand_loop:w
  }
```

(*End of definition for* `\text_expand:n` *and others. This function is documented on page 295.*)

`\text_declare_expand_equivalent:Nn`
`\text_declare_expand_equivalent:cn`

Create equivalents to allow replacement.

```
\cs_new_protected:Npn \text_declare_expand_equivalent:Nn #1#2
```

```
    {
      \tl_clear_new:c { l__text_expand_ \token_to_str:N #1 _tl }
      \tl_set:cn { l__text_expand_ \token_to_str:N #1 _tl } {#2}
    }
\cs_generate_variant:Nn \text_declare_expand_equivalent:Nn { c }
```

(*End of definition for* `\text_declare_expand_equivalent:Nn`. *This function is documented on page 295.*)

Prevent expansion of various standard values.

```
\tl_map_inline:nn
  { \` \' \^ \~ \= \u \. \" \r \H \v \d \c \k \b \t }
  { \text_declare_expand_equivalent:Nn #1 { \exp_not:n {#1} } }
\tl_map_inline:nn
  {
    \AA \aa
    \AE \ae
    \DH \dh
    \DJ \dj
    \IJ \ij
    \L  \l
    \NG \ng
    \O  \o
    \OE \oe
    \SS \ss
    \TH \th
  }
  { \text_declare_expand_equivalent:Nn #1 { \exp_not:n {#1} } }
⟨/package⟩
```

Chapter 88

l3text-case implementation

```
⟨*package⟩
⟨@@=text⟩
```

88.1 Case changing

\l_text_titlecase_check_letter_bool

Needed to determine the route used in titlecasing.

```
\bool_new:N \l_text_titlecase_check_letter_bool
\bool_set_true:N \l_text_titlecase_check_letter_bool
```

(*End of definition for* `\l_text_titlecase_check_letter_bool`. *This variable is documented on page 298.*)

\text_lowercase:n
\text_uppercase:n
\text_titlecase_all:n
\text_titlecase_first:n
\text_lowercase:nn
\text_uppercase:nn
\text_titlecase_all:nn
\text_titlecase_first:nn
__text_change_case:nnn

The user level functions here are all wrappers around the internal functions for case changing.

```
\cs_new:Npn \text_lowercase:n #1
  { \__text_change_case:nnn { lower } { } {#1} }
\cs_new:Npn \text_uppercase:n #1
  { \__text_change_case:nnn { upper } { } {#1} }
\cs_new:Npn \text_titlecase_all:n #1
  { \__text_change_case:nnn { title } { } {#1} }
\cs_new:Npn \text_titlecase_first:n #1
  { \__text_change_case:nnnn { title } { break } { } {#1} }
\cs_new:Npn \text_lowercase:nn #1#2
  { \__text_change_case:nnn { lower } {#1} {#2} }
\cs_new:Npn \text_uppercase:nn #1#2
  { \__text_change_case:nnn { upper } {#1} {#2} }
\cs_new:Npn \text_titlecase_all:nn #1#2
  { \__text_change_case:nnn { title } {#1} {#2} }
\cs_new:Npn \text_titlecase_first:nn #1#2
  { \__text_change_case:nnnn { title } { break } {#1} {#2} }
\cs_new:Npn \__text_change_case:nnn #1#2#3
  { \__text_change_case:nnnn {#1} {#1} {#2} {#3} }
```

(*End of definition for* `\text_lowercase:n` *and others. These functions are documented on page 296.*)

__text_change_case:nnnn
__text_change_case_auxi:nnnn
__text_change_case_auxii:nnnn
__text_change_case_BCP:nnnn
__text_change_case_BCP:nnnw
__text_change_case_BCP:nnnnnw
__text_change_case_store:n
__text_change_case_store:o
__text_change_case_store:V
__text_change_case_store:v
__text_change_case_store:e
__text_change_case_store:nw
__text_change_case_result:n

As for the expansion code, the business end of case changing is the handling of N-type tokens. First, we expand the input fully (so the loops here don't need to worry about awkward look-aheads and the like). Then we split into the different paths.

The code here needs to be f-type expandable to deal with the situation where case changing is applied in running text. There, we might have case changing as a document command and the text containing other non-expandable document commands.

```
\cs_set_eq:NN \MakeLowercase \text_lowercase
...
\MakeLowercase{\enquote*{A} text}
```

If we use an e-type expansion and wrap each token in `\exp_not:n`, that would explode: the document command grabs `\exp_not:n` as an argument, and things go badly wrong. So we have to wrap the entire result in exactly one `\exp_not:n`, or rather in the kernel version.

```
\cs_new:Npn \__text_change_case:nnnn #1#2#3#4
  {
    \__kernel_exp_not:w \exp_after:wN
      {
        \exp:w
        \exp_args:Ne \__text_change_case_auxi:nnnn
          { \text_expand:n {#4} }
          {#1} {#2} {#3}
      }
  }
\cs_new:Npn \__text_change_case_auxi:nnnn #1#2#3#4
  {
    \exp_args:No \__text_change_case_BCP:nnnn
      { \tl_to_str:n {#4} } {#1} {#2} {#3}
  }
\cs_new:Npe \__text_change_case_BCP:nnnn #1#2#3#4
  {
    \exp_not:N \__text_change_case_BCP:nnnw
      {#2} {#3} {#4} #1 \tl_to_str:n { -x- -x- } \exp_not:N \q__text_stop
  }
\use:e
  {
    \cs_new:Npn \exp_not:N \__text_change_case_BCP:nnnw
      #1#2#3#4 \tl_to_str:n { -x- } #5 \tl_to_str:n { -x- } #6
      \exp_not:N \q__text_stop
  }
  { \__text_change_case_BCP:nnnnnw {#1} {#2} {#3} {#5} {#4} #4 - \q__text_stop }
\cs_new:Npn \__text_change_case_BCP:nnnnnw #1#2#3#4#5#6 - #7 \q__text_stop
  {
    \bool_lazy_or:nnTF
      { \cs_if_exist_p:c { __text_change_case_ #2 _ #6 -x- #4 :nnnnn } }
      { \tl_if_exist_p:c { l__text_ #2 case_special_ #6 -x- #4 _tl } }
      { \__text_change_case_auxii:nnnn {#1} {#2} {#3} { #6 -x- #4 } }
      {
        \cs_if_exist:cTF { __text_change_case_ #2 _ #6 :nnnnn }
          { \__text_change_case_auxii:nnnn {#1} {#2} {#3} {#6} }
          { \__text_change_case_auxii:nnnn {#1} {#2} {#3} {#5} }
      }
  }
\cs_new:Npn \__text_change_case_auxii:nnnn #1#2#3#4
  {
    \group_align_safe_begin:
```

```
    \cs_if_exist_use:c { __text_change_case_boundary_ #2 _ #4 :Nnnnw }
    \__text_change_case_loop:nnnw {#2} {#3} {#4} #1
      \q__text_recursion_tail \q__text_recursion_stop
    \__text_change_case_result:n { }
  }
```

As for expansion, collect up the tokens for future use.

```
\cs_new:Npn \__text_change_case_store:n #1
  { \__text_change_case_store:nw {#1} }
\cs_generate_variant:Nn \__text_change_case_store:n { o , e , V , v }
\cs_new:Npn \__text_change_case_store:nw #1#2 \__text_change_case_result:n #3
  { #2 \__text_change_case_result:n { #3 #1 } }
\cs_new:Npn \__text_change_case_end:w #1 \__text_change_case_result:n #2
  {
    \group_align_safe_end:
    \exp_end:
    #2
  }
```

The main loop is the standard `tl action` type.

```
\cs_new:Npn \__text_change_case_loop:nnnw #1#2#3#4 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#4}
      { \__text_change_case_N_type:nnnN }
      {
        \tl_if_head_is_group:nTF {#4}
          { \use:c { __text_change_case_group_ #1 :nnnn } }
          { \__text_change_case_space:nnnw }
      }
    {#1} {#2} {#3} #4 \q__text_recursion_stop
  }
\cs_new:Npn \__text_change_case_break:w
  { \__text_change_case_break_aux:w \prg_do_nothing: }
\cs_new:Npn \__text_change_case_break_aux:w
  #1 \q__text_recursion_tail \q__text_recursion_stop
  {
    \__text_change_case_store:o {#1}
    \__text_change_case_end:w
  }
```

For a group, we *could* worry about whether this contains a character or not. However, that would make life very complex for little gain: exactly what a first character is is rather weakly-defined anyway. So if there is a group, we simply assume that a character has been seen, and for title case we switch to the "rest of the tokens" situation. To avoid having too much testing, we use a two-step process here to allow the titlecase functions to be separate.

```
\cs_new:Npn \__text_change_case_group_lower:nnnn #1#2#3#4
  {
    \__text_change_case_store:o
      {
        \exp_after:wN
          {
            \exp:w
            \__text_change_case_auxii:nnnn {#4} {#1} {#2} {#3}
          }
```

```
      }
    \__text_change_case_loop:nnnw {#1} {#2} {#3}
  }
\cs_new_eq:NN \__text_change_case_group_upper:nnnn
  \__text_change_case_group_lower:nnnn
\cs_new:Npn \__text_change_case_group_title:nnnn #1#2#3#4
  {
    \__text_change_case_store:o
      {
        \exp_after:wN
          {
            \exp:w
            \__text_change_case_auxii:nnnn {#4} {#1} {#2} {#3}
          }
      }
    \__text_change_case_skip:nnw {#2} {#3}
  }
\use:e
  {
    \cs_new:Npn \exp_not:N \__text_change_case_space:nnnw #1#2#3 \c_space_tl
  }
  {
    \__text_change_case_store:n { ~ }
    \cs_if_exist_use:cF { __text_change_case_space_ #2 :nnn }
      {
        \cs_if_exist_use:c { __text_change_case_boundary_ #1 _ #3 :Nnnnw }
        \__text_change_case_loop:nnnw
      }
        {#2} {#2} {#3}
  }
\cs_new:Npn \__text_change_case_space_break:nnn #1#2#3
  { \__text_change_case_break:w }
```

The first step of handling `N`-type tokens is to filter out the end-of-loop. That has to be done separately from the first real step as otherwise we pick up the wrong delimiter. The loop here is the same as the `expand` one, just passing the additional data long. If no close-math token is found then the final clean-up is forced (i.e. there is no assumption of "well-behaved" input in terms of math mode).

```
\cs_new:Npn \__text_change_case_N_type:nnnN #1#2#3#4
  {
    \__text_if_q_recursion_tail_stop_do:Nn #4
      { \__text_change_case_end:w }
    \__text_change_case_N_type_aux:nnnN {#1} {#2} {#3} #4
  }
\cs_new:Npn \__text_change_case_N_type_aux:nnnN #1#2#3#4
  {
    \exp_args:NV \__text_change_case_N_type:nnnnN
      \l_text_math_delims_tl {#1} {#2} {#3} #4
  }
\cs_new:Npn \__text_change_case_N_type:nnnnN #1#2#3#4#5
  {
    \__text_change_case_math_search:nnnNNN {#2} {#3} {#4} #5 #1
      \q__text_recursion_tail \q__text_recursion_tail
      \q__text_recursion_stop
```

```
  }
\cs_new:Npn \__text_change_case_math_search:nnnNNN #1#2#3#4#5#6
  {
    \__text_if_q_recursion_tail_stop_do:Nn #5
      { \__text_change_case_cs_check:nnnN {#1} {#2} {#3} #4 }
    \token_if_eq_meaning:NNTF #4 #5
      {
        \__text_use_i_delimit_by_q_recursion_stop:nw
          {
            \__text_change_case_store:n {#4}
            \__text_change_case_math_loop:nnnNw {#1} {#2} {#3} #6
          }
      }
      { \__text_change_case_math_search:nnnNNN {#1} {#2} {#3} #4 }
  }
\cs_new:Npn \__text_change_case_math_loop:nnnNw #1#2#3#4#5 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#5}
      { \__text_change_case_math_N_type:nnnNN }
      {
        \tl_if_head_is_group:nTF {#5}
          { \__text_change_case_math_group:nnnNn }
          { \__text_change_case_math_space:nnnNw }
      }
    {#1} {#2} {#3} #4 #5 \q__text_recursion_stop
  }
\cs_new:Npn \__text_change_case_math_N_type:nnnNN #1#2#3#4#5
  {
    \__text_if_q_recursion_tail_stop_do:Nn #5
      { \__text_change_case_end:w }
    \__text_change_case_store:n {#5}
    \token_if_eq_meaning:NNTF #5 #4
      { \__text_change_case_loop:nnnw {#1} {#2} {#3} }
      { \__text_change_case_math_loop:nnnNw {#1} {#2} {#3} #4 }
  }
\cs_new:Npn \__text_change_case_math_group:nnnNn #1#2#3#4#5
  {
    \__text_change_case_store:n { {#5} }
    \__text_change_case_math_loop:nnnNw {#1} {#2} {#3} #4
  }
\use:e
  {
    \cs_new:Npn \exp_not:N \__text_change_case_math_space:nnnNw #1#2#3#4
      \c_space_tl
  }
  {
    \__text_change_case_store:n { ~ }
    \__text_change_case_math_loop:nnnNw {#1} {#2} {#3} #4
  }
```

Once potential math-mode cases are filtered out the next stage is to test if the token grabbed is a control sequence: the two routes the code may take are then very different.

```
\cs_new:Npn \__text_change_case_cs_check:nnnN #1#2#3#4
  {
```

```
    \token_if_cs:NTF #4
      { \__text_change_case_exclude:nnnN {#1} {#2} {#3} }
      {
        \__text_codepoint_process:nN
          { \use:c { __text_change_case_custom_ #1 :nnnn } {#1} {#2} {#3} }
      }
        #4
  }
```

To deal with a control sequence there is first a need to test if it is on the list which indicate that case changing should be skipped. That's done using a loop as for the other special cases. If a hit is found then the argument is grabbed and passed through as-is.

```
\cs_new:Npn \__text_change_case_exclude:nnnN #1#2#3#4
  {
    \exp_args:Ne \__text_change_case_exclude:nnnnN
      {
        \exp_not:V \l_text_math_arg_tl
        \exp_not:V \l_text_case_exclude_arg_tl
      }
      {#1} {#2} {#3} #4
  }
\cs_new:Npn \__text_change_case_exclude:nnnnN #1#2#3#4#5
  {
    \__text_change_case_exclude:nnnNN {#2} {#3} {#4} #5 #1
      \q__text_recursion_tail \q__text_recursion_stop
  }
\cs_new:Npn \__text_change_case_exclude:nnnNN #1#2#3#4#5
  {
    \__text_if_q_recursion_tail_stop_do:Nn #5
      { \__text_change_case_replace:nnnN {#1} {#2} {#3} #4 }
    \str_if_eq:nnTF {#4} {#5}
      {
        \__text_use_i_delimit_by_q_recursion_stop:nw
          { \__text_change_case_exclude:nnnNw {#1} {#2} {#3} #4 }
      }
      { \__text_change_case_exclude:nnnNN {#1} {#2} {#3} #4 }
  }
\cs_new:Npn \__text_change_case_exclude:nnnNw #1#2#3#4#5#
  { \__text_change_case_exclude:nnnNnn {#1} {#2} {#3} {#4} {#5} }
\cs_new:Npn \__text_change_case_exclude:nnnNnn #1#2#3#4#5#6
  {
    \tl_if_blank:nTF {#5}
      { \__text_change_case_store:n { #4 {#6} } }
      {
        \__text_change_case_store:o
          {
            \exp_after:wN #4
              \exp:w \__text_change_case_auxii:nnnn {#5} {#1} {#2} {#3}
              {#6}
          }
      }
    \__text_change_case_loop:nnnw {#1} {#2} {#3}
  }
```

Deal with any specialist replacement for case changing.

```
\cs_new:Npn \__text_change_case_replace:nnnN #1#2#3#4
  {
    \cs_if_exist:cTF { l__text_case_ \token_to_str:N #4 _tl }
      {
        \__text_change_case_replace:vnnn
          { l__text_case_ \token_to_str:N #4 _tl } {#1} {#2} {#3}
      }
      { \__text_change_case_switch:nnnN {#1} {#2} {#3} #4 }
  }
\cs_new:Npn \__text_change_case_replace:nnnn #1#2#3#4
  { \__text_change_case_loop:nnnw {#2} {#3} {#4} #1 }
\cs_generate_variant:Nn \__text_change_case_replace:nnnn { v }
```

Allow for manually-controlled case switching.

```
\cs_new:Npn \__text_change_case_switch:nnnN #1#2#3#4
  {
    \cs_if_eq:NNTF #4 \text_case_switch:nnnn
      { \use:c { __text_change_case_switch_ #1 :nnnNnnnn  } }
      { \use:c { __text_change_case_letterlike_ #1 :nnnN } }
        {#1} {#2} {#3} #4
  }
\cs_new:Npn \__text_change_case_switch_lower:nnnNnnnn #1#2#3#4#5#6#7#8
  {
    \__text_change_case_store:n {#7}
    \__text_change_case_loop:nnnw {#1} {#2} {#3}
  }
\cs_new:Npn \__text_change_case_switch_upper:nnnNnnnn #1#2#3#4#5#6#7#8
  {
    \__text_change_case_store:n {#6}
    \__text_change_case_loop:nnnw {#1} {#2} {#3}
  }
\cs_new:Npn \__text_change_case_switch_title:nnnNnnnn #1#2#3#4#5#6#7#8
  {
    \__text_change_case_store:n {#8}
    \__text_change_case_skip:nnw {#2} {#3}
  }
```

Skip over material quickly after titlecase-first-only initials

```
\cs_new:Npn \__text_change_case_skip:nnw #1#2#3 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#3}
      { \__text_change_case_skip_N_type:nnN }
      {
        \tl_if_head_is_group:nTF {#3}
          { \__text_change_case_skip_group:nnn }
          { \__text_change_case_skip_space:nnw }
      }
        {#1} {#2} #3 \q__text_recursion_stop
  }
\cs_new:Npn \__text_change_case_skip_N_type:nnN #1#2#3
  {
    \__text_if_q_recursion_tail_stop_do:Nn #3
      { \__text_change_case_end:w }
    \__text_change_case_store:n {#3}
    \__text_change_case_skip:nnw {#1} {#2}
```

```
  }
\cs_new:Npn \__text_change_case_skip_group:nnn #1#2#3
  {
    \__text_change_case_store:n { {#3} }
    \__text_change_case_skip:nnw {#1} {#2}
  }
\cs_new:Npn \__text_change_case_skip_space:nnw #1#2
  { \__text_change_case_space:nnnw {#1} {#1} {#2} }
```

Letter-like commands may still be present: they are set up using a simple lookup approach, so can easily be handled with no loop. If there is no hit, we are at the end of the process: we loop around. Letter-like chars are all available only in upper- and lowercase, so titlecasing maps to the uppercase version.

```
\cs_new:Npn \__text_change_case_letterlike_lower:nnnN #1#2#3#4
  { \__text_change_case_letterlike:nnnnnN {#1} {#1} {#1} {#2} {#3} #4 }
\cs_new_eq:NN \__text_change_case_letterlike_upper:nnnN
  \__text_change_case_letterlike_lower:nnnN
\cs_new:Npn \__text_change_case_letterlike_title:nnnN #1#2#3#4
  { \__text_change_case_letterlike:nnnnnN { upper } { end } {#1} {#2} {#3} #4 }
\cs_new:Npn \__text_change_case_letterlike:nnnnnN #1#2#3#4#5#6
  {
    \cs_if_exist:cTF { c__text_ #1 case_ \token_to_str:N #6 _tl }
      {
        \__text_change_case_store:v
          { c__text_ #1 case_ \token_to_str:N #6 _tl }
        \use:c { __text_change_case_next_ #2 :nnn } {#2} {#4} {#5}
      }
      {
        \__text_change_case_store:n {#6}
        \cs_if_exist:cTF
          {
            c__text_
            \str_if_eq:nnTF {#1} { lower } { upper } { lower }
            case_ \token_to_str:N #6 _tl
          }
          { \use:c { __text_change_case_next_ #2 :nnn } {#2} {#4} {#5} }
          { \__text_change_case_loop:nnnw {#3} {#4} {#5} }
      }
  }
```

Check for a customised codepoint result.

```
\cs_new:Npn \__text_change_case_custom_lower:nnnn #1#2#3#4
  {
    \__text_change_case_custom:nnnnnn {#1} {#1} {#2} {#3} {#4}
      { \use:c { __text_change_case_codepoint_ #1 :nnnn } {#1} {#2} {#3} {#4} }
  }
\cs_new_eq:NN \__text_change_case_custom_upper:nnnn
  \__text_change_case_custom_lower:nnnn
\cs_new:Npn \__text_change_case_custom_title:nnnn #1#2#3#4
  {
    \__text_change_case_custom:nnnnnn { title } {#1} {#2} {#3} {#4}
      {
        \__text_change_case_custom:nnnnnn { upper } {#1} {#2} {#3} {#4}
          { \use:c { __text_change_case_codepoint_ #1 :nnnn } {#1} {#2} {#3} {#4} }
      }
```

```
  }
\cs_new:Npn \__text_change_case_custom:nnnnnn #1#2#3#4#5#6
  {
    \tl_if_exist:cTF { l__text_ #1 case _ \tl_to_str:n {#5} _ #4 _tl }
      {
        \__text_change_case_replace:vnnn
          { l__text_ #1 case _ \tl_to_str:n {#5} _ #4 _tl } {#2} {#3} {#4}
      }
      {
        \tl_if_exist:cTF { l__text_ #1 case _ \tl_to_str:n {#5} _tl }
          {
            \__text_change_case_replace:vnnn
              { l__text_ #1 case _ \tl_to_str:n {#5} _tl } {#2} {#3} {#4}
          }
          {#6}
      }
  }
```

For upper- and lowercase changes, once we get to this stage there are only a couple of questions remaining: is there a language-specific mapping and is there the special case of a terminal sigma. If not, then we pass to a simple codepoint mapping.

```
\cs_new:Npn \__text_change_case_codepoint_lower:nnnn #1#2#3#4
  {
    \cs_if_exist_use:cF { __text_change_case_lower_ #3 :nnnnn }
      { \__text_change_case_lower_sigma:nnnnn }
        {#1} {#1} {#2} {#3} {#4}
  }
\cs_new:Npn \__text_change_case_codepoint_upper:nnnn #1#2#3#4
  {
    \cs_if_exist_use:cF { __text_change_case_upper_ #3 :nnnnn }
      { \__text_change_case_codepoint:nnnnn }
        {#1} {#1} {#2} {#3} {#4}
  }
```

If the current character is an uppercase sigma, the a check is made on the next item in the input. If it is N-type and not a control sequence then there is a look-ahead phase: the logic here is simply based on letters or actives (to cover 8-bit engines).

```
\cs_new:Npn \__text_change_case_lower_sigma:nnnnn #1#2#3#4#5
  {
    \__text_codepoint_compare:nNnTF {#5} = { "03A3 }
      { \__text_change_case_lower_sigma:nnnnw {#2} }
      { \__text_change_case_codepoint:nnnnn {#1} {#2} }
        {#3} {#4} {#5}
  }
\cs_new:Npn \__text_change_case_lower_sigma:nnnnw #1#2#3#4#5 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#5}
      { \__text_change_case_lower_sigma:nnnnN {#4} }
      {
        \__text_change_case_store:e
          { \codepoint_generate:nn { "03C2 } { \__text_char_catcode:N #4 } }
        \__text_change_case_loop:nnnw
      }
        {#1} {#2} {#3} #5 \q__text_recursion_stop
  }
```

```
\cs_new:Npn \__text_change_case_lower_sigma:nnnnN #1#2#3#4#5
  {
    \__text_change_case_store:e
      {
        \bool_lazy_or:nnTF
          { \token_if_letter_p:N #5 }
          {
            \bool_lazy_and_p:nn
              { \token_if_active_p:N #5 }
              { \int_compare_p:nNn {`#5} > { "80 } }
          }
          { \codepoint_generate:nn { "03C3 } { \__text_char_catcode:N #1 } }
          { \codepoint_generate:nn { "03C2 } { \__text_char_catcode:N #1 } }
      }
    \__text_change_case_loop:nnnw {#2} {#3} {#4} #5
  }
```

For titlecasing, we need to obtain the general category of the current codepoint.

```
\cs_new:Npn \__text_change_case_codepoint_title:nnnn #1#2#3#4
  {
    \bool_if:NTF \l_text_titlecase_check_letter_bool
      {
        \exp_args:Ne \__text_change_case_codepoint_title_auxi:nnnn
          {
            \codepoint_to_category:n
              { \__text_codepoint_from_chars:Nw #4 }
          }
      }
      { \__text_change_case_codepoint_title:nnn }
        {#2} {#3} {#4}
  }
\cs_new:Npn \__text_change_case_codepoint_title_auxi:nnnn #1#2#3#4
  {
    \tl_if_head_eq_charcode:nNTF {#1} { L }
      { \__text_change_case_codepoint_title:nnn }
      { \__text_change_case_codepoint_title_auxii:nnnn { title } }
        {#2} {#3} {#4}
  }
\cs_new:Npn \__text_change_case_codepoint_title:nnn #1#2#3
  { \__text_change_case_codepoint_title_auxii:nnnn { end } {#1} {#2} {#3} }
\cs_new:Npn \__text_change_case_codepoint_title_auxii:nnnn #1#2#3#4
  {
    \cs_if_exist_use:cF { __text_change_case_title_ #3 :nnnnn }
      {
        \cs_if_exist_use:cF { __text_change_case_upper_ #3 :nnnnn }
          { \__text_change_case_codepoint:nnnnn }
      }
        { title } {#1} {#2} {#3} {#4}
  }
\cs_new:Npn \__text_change_case_codepoint:nnnnn #1#2#3#4#5
  {
    \bool_lazy_and:nnTF
      { \tl_if_single_p:n {#5} }
      { \token_if_active_p:N #5 }
      { \__text_change_case_store:n {#5} }
```

```
      {
        \__text_change_case_store:e
          { \__text_change_case_codepoint:nn {#1} {#5} }
      }
    \use:c { __text_change_case_next_ #2 :nnn } {#2} {#3} {#4}
  }
\cs_new:Npn \__text_change_case_codepoint:nn #1#2
  {
    \__text_change_case_codepoint:fnn
      { \int_eval:n { \__text_codepoint_from_chars:Nw #2 } } {#1} {#2}
  }
\cs_new:Npn \__text_change_case_codepoint:nnn #1#2#3
  {
    \exp_args:Ne \__text_change_case_codepoint_aux:nn
      { \__kernel_codepoint_case:nn { #2 case } {#1} } {#3}
  }
\cs_generate_variant:Nn \__text_change_case_codepoint:nnn { f }
```

Avoid high chars with pTEX.

```
\sys_if_engine_ptex:T
  {
    \cs_new_eq:NN \__text_change_case_codepoint_aux:nnn
      \__text_change_case_codepoint:nnn
    \cs_gset:Npn \__text_change_case_codepoint:nnn #1#2#3
      {
        \int_compare:nNnTF {#1} = { -1 }
          { \exp_not:n {#3} }
          { \__text_change_case_codepoint_aux:nnn {#1} {#2} {#3} }
      }
  }
\cs_new:Npn \__text_change_case_codepoint_aux:nn #1#2
  {
    \use:e { \__text_change_case_codepoint_aux:nnnn #1 {#2} }
  }
\cs_new:Npn \__text_change_case_codepoint_aux:nnnn #1#2#3#4
  {
    \__text_codepoint_compare:nNnTF {#4} = {#1}
      { \exp_not:n {#4} }
      {
        \codepoint_generate:nn {#1}
          { \__text_change_case_catcode:nn {#4} {#1} }
        \tl_if_blank:nF {#2}
          {
            \codepoint_generate:nn {#2}
              { \char_value_catcode:n {#2} }
            \tl_if_blank:nF {#3}
              {
                \codepoint_generate:nn {#3}
                  { \char_value_catcode:n {#3} }
              }
          }
      }
  }
```

We need to ensure that only valid catcode-extraction is attempted. That's fine with

Unicode engines but needs a bit of work with 8-bit ones. The logic is that if the original codepoint was in the ASCII range, we keep the catcode. Otherwise, if the target is in the ASCII range, we use the standard catcode. If neither are true, we set as 13 on the grounds that this will be what is used anyway!

```
\bool_lazy_or:nnTF
  { \sys_if_engine_luatex_p: }
  { \sys_if_engine_xetex_p: }
  {
    \cs_new:Npn \__text_change_case_catcode:nn #1#2
      { \__text_char_catcode:N #1 }
  }
  {
    \cs_new:Npn \__text_change_case_catcode:nn #1#2
      {
        \__text_codepoint_compare:nNnTF {#1} < { "80 }
          { \__text_char_catcode:N #1 }
          {
            \int_compare:nNnTF {#2} < { "80 }
              { \char_value_catcode:n {#2} }
              { 13 }
          }
      }
  }
\cs_new:Npn \__text_change_case_next_lower:nnn #1#2#3
  { \__text_change_case_loop:nnnw {#1} {#2} {#3} }
\cs_new_eq:NN \__text_change_case_next_upper:nnn
  \__text_change_case_next_lower:nnn
\cs_new_eq:NN \__text_change_case_next_title:nnn
  \__text_change_case_next_lower:nnn
\cs_new:Npn \__text_change_case_next_end:nnn #1#2#3
  { \__text_change_case_skip:nnw {#2} {#3} }
```

(*End of definition for* `\__text_change_case:nnnn` *and others.*)

`\text_declare_case_equivalent:Nn`

Create equivalents to allow replacement.

```
\cs_new_protected:Npn \text_declare_case_equivalent:Nn #1#2
  {
    \tl_clear_new:c { l__text_case_ \token_to_str:N #1 _tl }
    \tl_set:cn { l__text_case_ \token_to_str:N #1 _tl } {#2}
  }
```

(*End of definition for* `\text_declare_case_equivalent:Nn`*. This function is documented on page* *297.*)

`\text_declare_lowercase_mapping:nn`
`\text_declare_titlecase_mapping:nn`
`\text_declare_uppercase_mapping:nn`
`\__text_declare_case_mapping:nnn`
`\__text_declare_case_mapping_aux:nnn`
`\text_declare_lowercase_mapping:nnn`
`\text_declare_titlecase_mapping:nnn`
`\text_declare_uppercase_mapping:nnn`
`\__text_declare_case_mapping:nnnn`
`\__text_declare_case_mapping_aux:nnnn`

Codepoint customisation.

```
\cs_new_protected:Npn \text_declare_lowercase_mapping:nn #1#2
  { \__text_declare_case_mapping:nnn { lower } {#1} {#2} }
\cs_new_protected:Npn \text_declare_titlecase_mapping:nn #1#2
  { \__text_declare_case_mapping:nnn { title } {#1} {#2} }
\cs_new_protected:Npn \text_declare_uppercase_mapping:nn #1#2
  { \__text_declare_case_mapping:nnn { upper } {#1} {#2} }
\cs_new_protected:Npn \__text_declare_case_mapping:nnn #1#2#3
  {
    \exp_args:Ne \__text_declare_case_mapping_aux:nnn
      { \codepoint_str_generate:n {#2} } {#1} {#3}
```

```
  }
\cs_new_protected:Npn \__text_declare_case_mapping_aux:nnn #1#2#3
  {
    \tl_clear_new:c { l__text_ #2 case _ #1 _tl }
    \tl_set:cn { l__text_ #2 case _ #1 _ tl } {#3}
  }
\cs_new_protected:Npn \text_declare_lowercase_mapping:nnn #1#2#3
  { \__text_declare_case_mapping:nnnn { lower } {#1} {#2} {#3} }
\cs_new_protected:Npn \text_declare_titlecase_mapping:nnn #1#2#3
  { \__text_declare_case_mapping:nnnn { title } {#1} {#2} {#3} }
\cs_new_protected:Npn \text_declare_uppercase_mapping:nnn #1#2#3
  { \__text_declare_case_mapping:nnnn { upper } {#1} {#2} {#3} }
\cs_new_protected:Npn \__text_declare_case_mapping:nnnn #1#2#3#4
  {
    \exp_args:Ne \__text_declare_case_mapping_aux:nnnn
      { \codepoint_str_generate:n {#3} } {#1} {#2} {#4}
  }
\cs_new_protected:Npn \__text_declare_case_mapping_aux:nnnn #1#2#3#4
  {
    \tl_clear_new:c { l__text_ #2 case _ #1 _ #3 _tl }
    \tl_set:cn { l__text_ #2 case _ #1 _ #3 _ tl } {#4}
    \tl_clear_new:c { l__text_ #2 case_special_ #3 _tl }
  }
```

(*End of definition for* `\text_declare_lowercase_mapping:nn` *and others. These functions are documented on page* *297*.)

\text_case_switch:nnnn
__text_case_switch_marker:

Set up the mechanism for manual case switching.

```
\cs_new:Npn \text_case_switch:nnnn #1#2#3#4
  {
    \__text_case_switch_marker:
    #1
  }
\cs_new:Npn \__text_case_switch_marker: { }
```

(*End of definition for* `\text_case_switch:nnnn` *and* `\__text_case_switch_marker:`*. This function is documented on page* *297*.)

__text_change_case_generate:n

A utility.

```
\cs_new:Npn \__text_change_case_generate:n #1
  { \codepoint_generate:nn {#1} { \char_value_catcode:n {#1} } }
```

(*End of definition for* `\__text_change_case_generate:n`.)

__text_change_case_upper_de-x-eszett:nnnnn
__text_change_case_upper_de-alt:nnnnn

A simple alternative version for German.

```
\cs_new:cpn { __text_change_case_upper_de-x-eszett:nnnnn } #1#2#3#4#5
  {
    \__text_codepoint_compare:nNnTF {#5} = { "00DF }
      {
        \__text_change_case_store:e
          {
            \codepoint_generate:nn { "1E9E }
              { \__text_change_case_catcode:nn {#5} { "1E9E } }
          }
        \use:c { __text_change_case_next_ #2 :nnn }
```

```
            {#2} {#3} {#4}
        }
        { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
  }
\cs_new_eq:cc { __text_change_case_upper_de-alt:nnnnn }
  { __text_change_case_upper_de-x-eszett:nnnnn }
```

(*End of definition for* `\__text_change_case_upper_de-x-eszett:nnnnn` *and* `\__text_change_case_upper_de-alt:nnnnn`*.*)

__text_change_case_upper_el:nnnnn
__text_change_case_upper_el-x-iota:nnnnn
__text_change_case_upper_el_aux:nnnnn
__text_change_case_upper_el:nnnn
__text_change_case_upper_el:nnnnw
__text_change_case_upper_el:nnnnN
__text_change_case_upper_el_aux:nnnnN
_change_case_upper_el_ypogegrammeni:nnnnnnw
_change_case_upper_el_ypogegrammeni:nnnnnnN
_change_case_upper_el_ypogegrammeni:nnnnnnn
__text_change_case_upper_el_dialytika:nnnn
__text_change_case_upper_el_dialytika:n
__text_change_case_upper_el_hiatus:nnnnw
__text_change_case_upper_el_hiatus:nnnnN
__text_change_case_upper_el_hiatus:nnnnn
__text_change_case_upper_el_ypogegrammeni:n
change_case_upper_el-x-iota_ypogegrammeni:n
__text_change_case_upper_el_stress:nn
__text_change_case_upper_el_gobble:nnnw
__text_change_case_upper_el_gobble:nnnN
__text_change_case_upper_el_gobble:nnnn
__text_change_case_if_greek:n
__text_change_case_if_greek:n*TF*
xt_change_case_if_greek_spacing_diacritic:n
_change_case_if_greek_spacing_diacritic:n*TF*
__text_change_case_if_greek_accent:n
__text_change_case_if_greek_accent:n*TF*
__text_change_case_if_greek_breathing:n
__text_change_case_if_greek_breathing:n*TF*
__text_change_case_if_greek_stress:n
__text_change_case_if_greek_stress:n*TF*
__text_change_case_if_takes_dialytika:n
__text_change_case_if_takes_dialytika:n*TF*
__text_change_case_if_takes_ypogegrammeni:n
text_change_case_if_takes_ypogegrammeni:n*TF*

For Greek uppercasing, we need to know if characters *in the Greek range* have accents. That means doing a NFD conversion first, then starting a search. As described by the Unicode CLDR, Greek accents need to be found *after* any U+0308 (diaeresis) and are done in two groups to allow for the canonical ordering. The implementation here follows the data and examples from ICU (https://icu.unicode.org/design/case/greek-upper), although necessarily the implementation is somewhat different. The *ypogegrammeni* is filtered out here as it is not actually in the Greek range, so gets lost if we leave until later. The one Greek codepoint we skip is the numeral sign and question mark: the first has an awkward NFD for pdfTEX so is best left unchanged, and the latter has issues concerning how `LGR` outputs the input and output (differently!).

```
\cs_new:Npn \__text_change_case_upper_el:nnnnn #1#2#3#4#5
  {
    \bool_lazy_and:nnTF
      { \__text_change_case_if_greek_p:n {#5} }
      {
        ! \bool_lazy_or_p:nn
          { \__text_codepoint_compare_p:nNn {#5} = { "0374 } }
          { \__text_codepoint_compare_p:nNn {#5} = { "037E } }
      }
      {
        \__text_change_case_if_greek_spacing_diacritic:nTF {#5}
          {
            \__text_change_case_store:n {#5}
            \__text_change_case_loop:nnnw
          }
          {
            \exp_args:Ne \__text_change_case_upper_el:nnnn
              {
                \codepoint_to_nfd:n
                  { \__text_codepoint_from_chars:Nw #5 }
              }
          }
            {#2} {#3} {#4}
      }
      {
        \__text_codepoint_compare:nNnTF {#5} = { "0345 }
          {
            \__text_change_case_store:e
              {
                \codepoint_generate:nn { "0399 }
                  { \char_value_catcode:n { "0399 } }
              }
            \__text_change_case_loop:nnnw {#2} {#3} {#4}
          }
```

```
        { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
    }
  }
\cs_new_eq:cN { __text_change_case_upper_el-x-iota:nnnnn }
  \__text_change_case_upper_el:nnnnn
\cs_new:Npn \__text_change_case_upper_el:nnnn #1#2#3#4
  {
    \__text_codepoint_process:nN
      { \__text_change_case_upper_el:nnnnw {#2} {#3} {#4} } #1
  }
```

At this stage we have the first NFD codepoint as #3. What we need to know is whether after that we have another character, either from the NFD or directly in the input. If not, we store the changed character at this stage.

```
\cs_new:Npn \__text_change_case_upper_el:nnnnw #1#2#3#4#5 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#5}
      { \__text_change_case_upper_el:nnnnN {#4} }
      {
        \__text_change_case_store:e
          { \__text_change_case_codepoint:nn { upper } {#4} }
        \__text_change_case_loop:nnnw
      }
        {#1} {#2} {#3} #5 \q__text_recursion_stop
  }
```

Now, we check the detail of the next codepoint: again we filter out the not-a-char cases, before checking if it's an dialytika, accent or diacritic. (The latter do not have the same hiatus behavior as accents.) There is additional work if the codepoint can take a ypogegrammeni: there, we need to move any ypogegrammeni to after accents (in case the input is not normalised). The ypogegrammeni itself is handled separately.

```
\cs_new:Npn \__text_change_case_upper_el:nnnnN #1#2#3#4#5
  {
    \token_if_cs:NTF #5
      {
        \__text_change_case_store:e
          { \__text_change_case_codepoint:nn { upper } {#1} }
        \__text_change_case_loop:nnnw {#2} {#3} {#4} #5
      }
      {
        \__text_change_case_if_takes_ypogegrammeni:nTF {#1}
          {
            \__text_change_case_upper_el_ypogegrammeni:nnnnnnw
              {#1} {#2} {#3} {#4} { } { } #5
          }
          { \__text_change_case_upper_el_aux:nnnnN {#1} {#2} {#3} {#4} #5 }
      }
  }
\cs_new:Npn \__text_change_case_upper_el_ypogegrammeni:nnnnnnw
  #1#2#3#4#5#6#7 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#7}
      {
        \__text_change_case_upper_el_ypogegrammeni:nnnnnnN
```

```
        {#1} {#2} {#3} {#4} {#5} {#6}
    }
    { \__text_change_case_upper_el_aux:nnnnN {#1} {#2} {#3} {#4} #5#6 }
      #7 \q__text_recursion_stop
  }
\cs_new:Npn \__text_change_case_upper_el_ypogegrammeni:nnnnnnN #1#2#3#4#5#6#7
  {
    \token_if_cs:NTF #7
      { \__text_change_case_upper_el_aux:nnnnN {#1} {#2} {#3} {#4} #5#6 }
      {
        \__text_codepoint_process:nN
          {
            \__text_change_case_upper_el_ypogegrammeni:nnnnnnn
              {#1} {#2} {#3} {#4} {#5} {#6}
          }
      }
        #7
  }
\cs_new:Npn \__text_change_case_upper_el_ypogegrammeni:nnnnnnn #1#2#3#4#5#6#7
  {
    \__text_codepoint_compare:nNnTF {#7} = { "0345 }
      {
        \__text_change_case_upper_el_ypogegrammeni:nnnnnnw
          {#1} {#2} {#3} {#4} {#5} {#7}
      }
      {
        \bool_lazy_or:nnTF
          { \__text_change_case_if_greek_accent_p:n {#7} }
          { \__text_change_case_if_greek_breathing_p:n {#7} }
          {
            \__text_change_case_upper_el_ypogegrammeni:nnnnnnw
              {#1} {#2} {#3} {#4} {#5#7} {#6}
          }
          { \__text_change_case_upper_el_aux:nnnnN {#1} {#2} {#3} {#4} #5#6 #7 }
      }
  }
\cs_new:Npn \__text_change_case_upper_el_aux:nnnnN #1#2#3#4#5
  {
    \__text_codepoint_process:nN
      { \__text_change_case_upper_el_aux:nnnnn {#1} {#2} {#3} {#4} } #5
  }
\cs_new:Npn \__text_change_case_upper_el_aux:nnnnn #1#2#3#4#5
  {
    \__text_codepoint_compare:nNnTF {#5} = { "0308 }
      { \__text_change_case_upper_el_dialytika:nnnn {#2} {#3} {#4} {#1} }
      {
        \__text_change_case_if_greek_accent:nTF {#5}
          { \__text_change_case_upper_el_hiatus:nnnnw {#2} {#3} {#4} {#1} }
          {
            \__text_change_case_if_greek_breathing:nTF {#5}
              { \__text_change_case_upper_el:nnnn {#1} {#2} {#3} {#4} }
              {
                \__text_codepoint_compare:nNnTF {#5} = { "0345 }
                  {
```

```
                      \__text_change_case_store:e
                        { \use:c { __text_change_case_upper_ #4 _ypogegrammeni:n } {#1} }
                      \__text_change_case_loop:nnnw {#2} {#3} {#4}
                    }
                    {
                      \__text_change_case_if_greek_stress:nTF {#5}
                        {
                          \__text_change_case_store:e
                            { \__text_change_case_upper_el_stress:nn {#1} {#5} }
                          \__text_change_case_loop:nnnw {#2} {#3} {#4}

                        }
                        {
                          \__text_change_case_store:e
                            { \__text_change_case_codepoint:nn { upper } {#1} }
                          \__text_change_case_loop:nnnw {#2} {#3} {#4} #5
                        }
                    }
                }
            }
        }
    }
```

We handle *dialytika* in parts as it's also needed for the hiatus. We know only two letters take it, so we can shortcut here on the second part of the tests.

```
\cs_new:Npn \__text_change_case_upper_el_dialytika:nnnn #1#2#3#4
  {
    \__text_change_case_if_takes_dialytika:nTF {#4}
      { \__text_change_case_upper_el_dialytika:n {#4} }
      {
        \__text_change_case_store:e
          { \__text_change_case_codepoint:nn { upper } {#4} }
      }
    \__text_change_case_upper_el_gobble:nnnw {#1} {#2} {#3}
  }
\cs_new:Npn \__text_change_case_upper_el_dialytika:n #1
  {
    \__text_change_case_store:e
      {
        \bool_lazy_or:nnTF
          { \__text_codepoint_compare_p:nNn {#1} = { "0399 } }
          { \__text_codepoint_compare_p:nNn {#1} = { "03B9 } }
          {
            \codepoint_generate:nn { "03AA }
              { \__text_change_case_catcode:nn {#1} { "03AA } }
          }
          {
            \codepoint_generate:nn { "03AB }
              { \__text_change_case_catcode:nn {#1} { "03AB } }
          }
      }
  }
```

Adding a hiatus needs some of the same ideas, but if there is not one we skip this code point, hence needing a separate function.

```
\cs_new:Npn \__text_change_case_upper_el_hiatus:nnnnw
  #1#2#3#4#5 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#5}
      { \__text_change_case_upper_el_hiatus:nnnnN {#4} }
      {
        \__text_change_case_store:e
          { \__text_change_case_codepoint:nn { upper } {#4} }
        \__text_change_case_loop:nnnw
      }
        {#1} {#2} {#3} #5 \q__text_recursion_stop
  }
\cs_new:Npn \__text_change_case_upper_el_hiatus:nnnnN #1#2#3#4#5
  {
    \token_if_cs:NTF #5
      {
        \__text_change_case_store:e
          { \__text_change_case_codepoint:nn { upper } {#1} }
        \__text_change_case_loop:nnnw {#2} {#3} {#4} #5
      }
      {
        \__text_codepoint_process:nN
          { \__text_change_case_upper_el_hiatus:nnnnn {#1} {#2} {#3} {#4} } #5
      }
  }
\cs_new:Npn \__text_change_case_upper_el_hiatus:nnnnn #1#2#3#4#5
  {
    \__text_change_case_if_takes_dialytika:nTF {#5}
      {
        \__text_change_case_store:e
          { \__text_change_case_codepoint:nn { upper } {#1} }
        \__text_change_case_upper_el_dialytika:n {#5}
        \__text_change_case_upper_el_gobble:nnnw {#2} {#3} {#4}
      }
      { \__text_change_case_upper_el:nnnn {#1} {#2} {#3} {#4} #5 }
  }
```

Handling the *ypogegrammeni* output depends on the selected approach

```
\cs_new:Npn \__text_change_case_upper_el_ypogegrammeni:n #1
  {
    \exp_args:Ne \__text_change_case_generate:n
      {
        \int_case:nn
          { \__text_codepoint_from_chars:Nw #1 }
          {
            { "0391 } { "1FBC }
            { "03B1 } { "1FBC }
            { "0397 } { "1FCC }
            { "03B7 } { "1FCC }
            { "03A9 } { "1FFC }
            { "03C9 } { "1FFC }
          }
      }
  }
\cs_new:cpn { __text_change_case_upper_el-x-iota_ypogegrammeni:n } #1
```

```
  {
    \__text_change_case_codepoint:nn { upper } {#1}
    \codepoint_generate:nn { "0399 }
      { \char_value_catcode:n { "0399 } }
  }
```

We choose to retain stress diacritics, but we also need to recombine them for pdfTeX. That is handled here.

```
\cs_new:Npn \__text_change_case_upper_el_stress:nn #1#2
  {
    \exp_args:Ne \__text_change_case_generate:n
      {
        \int_case:nn
          { \__text_codepoint_from_chars:Nw #2 }
          {
            { "0304 }
              {
                \int_case:nn { \__text_codepoint_from_chars:Nw #1 }
                  {
                    { "0391 } { "1FB9 }
                    { "03B1 } { "1FB9 }
                    { "0399 } { "1FD9 }
                    { "03B9 } { "1FD9 }
                    { "03A5 } { "1FE9 }
                    { "03C5 } { "1FE9 }
                  }
              }
            { "0306 }
              {
                \int_case:nn { \__text_codepoint_from_chars:Nw #1 }
                  {
                    { "0391 } { "1FB8 }
                    { "03B1 } { "1FB8 }
                    { "0399 } { "1FD8 }
                    { "03B9 } { "1FD8 }
                    { "03A5 } { "1FE8 }
                    { "03C5 } { "1FE8 }
                  }
              }
          }
      }
  }
```

For clearing out trailing combining marks after we have dealt with the first one.

```
\cs_new:Npn \__text_change_case_upper_el_gobble:nnnw
  #1#2#3#4 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#4}
      { \__text_change_case_upper_el_gobble:nnnN }
      { \__text_change_case_loop:nnnw }
        {#1} {#2} {#3} #4 \q__text_recursion_stop
  }
\cs_new:Npn \__text_change_case_upper_el_gobble:nnnN #1#2#3#4
  {
    \token_if_cs:NTF #4
```

```
      { \__text_change_case_loop:nnnw {#1} {#2} {#3} }
      {
        \__text_codepoint_process:nN
          { \__text_change_case_upper_el_gobble:nnnn {#1} {#2} {#3} }
      }
        #4
  }
\cs_new:Npn \__text_change_case_upper_el_gobble:nnnn #1#2#3#4
  {
    \bool_lazy_or:nnTF
      { \__text_change_case_if_greek_accent_p:n {#4} }
      { \__text_change_case_if_greek_breathing_p:n {#4} }
      { \__text_change_case_upper_el_gobble:nnnw {#1} {#2} {#3} }
      { \__text_change_case_loop:nnnw {#1} {#2} {#3} #4 }
  }
```

Luckily the Greek range is limited and clear.

```
\prg_new_conditional:Npnn \__text_change_case_if_greek:n #1 { p , TF }
  {
    \exp_args:Nf \__text_change_case_if_greek:n
      { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
  }
\cs_new:Npn \__text_change_case_if_greek:n #1
  {
    \if_int_compare:w #1 < "0370 \exp_stop_f:
      \prg_return_false:
    \else:
      \if_int_compare:w #1 > "03FF \exp_stop_f:
        \if_int_compare:w #1 < "1F00 \exp_stop_f:
          \prg_return_false:
        \else:
          \if_int_compare:w #1 > "1FFF \exp_stop_f:
            \if_int_compare:w #1 = "2126 \exp_stop_f:
              \prg_return_true:
            \else:
              \prg_return_false:
            \fi:
          \else:
            \prg_return_true:
          \fi:
        \fi:
      \else:
        \prg_return_true:
      \fi:
    \fi:
  }
```

We follow ICU in adding a few extras to the accent list here.

```
\prg_new_conditional:Npnn \__text_change_case_if_greek_accent:n #1 { TF , p }
  {
    \exp_args:Nf \__text_change_case_if_greek_accent:n
      { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
  }
\cs_new:Npn \__text_change_case_if_greek_accent:n #1
  {
```

```
    \if_int_compare:w #1 = "0300 \exp_stop_f:
      \prg_return_true:
    \else:
      \if_int_compare:w #1 = "0301 \exp_stop_f:
        \prg_return_true:
      \else:
        \if_int_compare:w #1 = "0342 \exp_stop_f:
          \prg_return_true:
        \else:
          \if_int_compare:w #1 = "0302 \exp_stop_f:
            \prg_return_true:
          \else:
            \if_int_compare:w #1 = "0303 \exp_stop_f:
              \prg_return_true:
            \else:
              \if_int_compare:w #1 = "0311 \exp_stop_f:
                \prg_return_true:
              \else:
                \prg_return_false:
              \fi:
            \fi:
          \fi:
        \fi:
      \fi:
    \fi:
  }
\prg_new_conditional:Npnn \__text_change_case_if_greek_spacing_diacritic:n
  #1 { TF }
  {
    \exp_args:Nf \__text_change_case_if_greek_spacing_diacritic:n
      { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
  }
\cs_new:Npn \__text_change_case_if_greek_spacing_diacritic:n #1
  {
    \if_int_compare:w #1 < "1FBD \exp_stop_f:
      \if_int_compare:w #1 = "037A \exp_stop_f:
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \else:
      \if_int_compare:w #1 = "1FBD \exp_stop_f:
        \prg_return_true:
      \else:
        \if_int_compare:w #1 = "1FBF \exp_stop_f:
          \prg_return_true:
        \else:
          \if_int_compare:w #1 = "1FC0 \exp_stop_f:
            \prg_return_true:
          \else:
            \if_int_compare:w #1 = "1FC1 \exp_stop_f:
              \prg_return_true:
            \else:
              \if_int_compare:w #1 = "1FCD \exp_stop_f:
```

```
                \prg_return_true:
              \else:
                \if_int_compare:w #1 = "1FCE \exp_stop_f:
                  \prg_return_true:
                \else:
                  \if_int_compare:w #1 = "1FCF \exp_stop_f:
                    \prg_return_true:
                  \else:
                    \if_int_compare:w #1 = "1FDD \exp_stop_f:
                      \prg_return_true:
                    \else:
                      \if_int_compare:w #1 = "1FDE \exp_stop_f:
                        \prg_return_true:
                      \else:
                        \if_int_compare:w #1 = "1FDF \exp_stop_f:
                          \prg_return_true:
                        \else:
                          \if_int_compare:w #1 = "1FED \exp_stop_f:
                            \prg_return_true:
                          \else:
                            \if_int_compare:w #1 = "1FEE \exp_stop_f:
                              \prg_return_true:
                            \else:
                              \if_int_compare:w #1 = "1FEF \exp_stop_f:
                                \prg_return_true:
                              \else:
                                \if_int_compare:w #1 = "1FFD \exp_stop_f:
                                  \prg_return_true:
                                \else:
                                  \if_int_compare:w #1 = "1FFE \exp_stop_f:
                                    \prg_return_true:
                                  \else:
                                    \prg_return_false:
                                  \fi:
                                \fi:
                              \fi:
                            \fi:
                          \fi:
                        \fi:
                      \fi:
                    \fi:
                  \fi:
                \fi:
              \fi:
            \fi:
          \fi:
        \fi:
      \fi:
    \fi:
  }
\prg_new_conditional:Npnn \__text_change_case_if_greek_breathing:n
  #1 { TF , p }
  {
    \exp_args:Nf \__text_change_case_if_greek_breathing:n
```

```
      { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
  }
\cs_new:Npn \__text_change_case_if_greek_breathing:n #1
  {
    \if_int_compare:w #1 = "0313 \exp_stop_f:
      \prg_return_true:
    \else:
      \if_int_compare:w #1 = "0314 \exp_stop_f:
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \fi:
  }
\prg_new_conditional:Npnn \__text_change_case_if_greek_stress:n
  #1 { TF , p }
  {
    \exp_args:Nf \__text_change_case_if_greek_stress:n
      { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
  }
\cs_new:Npn \__text_change_case_if_greek_stress:n #1
  {
    \if_int_compare:w #1 = "0304 \exp_stop_f:
      \prg_return_true:
    \else:
      \if_int_compare:w #1 = "0306 \exp_stop_f:
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \fi:
  }
\prg_new_conditional:Npnn \__text_change_case_if_takes_dialytika:n #1 { TF }
  {
    \exp_args:Nf \__text_change_case_if_takes_dialytika:n
      { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
  }
\cs_new:Npn \__text_change_case_if_takes_dialytika:n #1
  {
    \if_int_compare:w #1 = "0399 \exp_stop_f:
      \prg_return_true:
    \else:
      \if_int_compare:w #1 = "03B9 \exp_stop_f:
        \prg_return_true:
      \else:
        \if_int_compare:w #1 = "03A5 \exp_stop_f:
          \prg_return_true:
        \else:
          \if_int_compare:w #1 = "03C5 \exp_stop_f:
            \prg_return_true:
          \else:
            \prg_return_false:
          \fi:
        \fi:
```

```
      \fi:
    \fi:
  }
\prg_new_conditional:Npnn \__text_change_case_if_takes_ypogegrammeni:n #1 { TF }
  {
    \exp_args:Nf \__text_change_case_if_takes_ypogegrammeni:n
      { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
  }
\cs_new:Npn \__text_change_case_if_takes_ypogegrammeni:n #1
  {
    \if_int_compare:w #1 = "03B1 \exp_stop_f:
      \prg_return_true:
    \else:
      \if_int_compare:w #1 = "03B7 \exp_stop_f:
        \prg_return_true:
      \else:
        \if_int_compare:w #1 = "03C9 \exp_stop_f:
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      \fi:
    \fi:
  }
```

(*End of definition for* `\__text_change_case_upper_el:nnnnn` *and others.*)

__text_change_case_boundary_upper_el:Nnnnw
_change_case_boundary_upper_el-x-iota:Nnnnw
__text_change_case_boundary_upper_el:nnnN
__text_change_case_boundary_upper_el:nnnn
__text_change_case_boundary_upper_el:nnnnw

There is one things that need special treatment at the start of words in Greek. For an isolated accent *eta*, which is handled by seeing if we have exactly one of the affected codepoints followed by a space or brace group.

```
\cs_new:Npn \__text_change_case_boundary_upper_el:Nnnnw
  #1#2#3#4#5 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#5}
      { \__text_change_case_boundary_upper_el:nnnN }
      { \__text_change_case_loop:nnnw }
        {#2} {#3} {#4} #5 \q__text_recursion_stop
  }
\cs_new_eq:cN { __text_change_case_boundary_upper_el-x-iota:Nnnnw }
  \__text_change_case_boundary_upper_el:Nnnnw
\cs_new:Npn \__text_change_case_boundary_upper_el:nnnN #1#2#3#4
  {
    \token_if_cs:NTF #4
      { \__text_change_case_loop:nnnw {#1} {#2} {#3} }
      {
        \__text_codepoint_process:nN
          { \__text_change_case_boundary_upper_el:nnnn {#1} {#2} {#3} }
      }
        #4
  }
\cs_new:Npn \__text_change_case_boundary_upper_el:nnnn #1#2#3#4
  {
    \bool_lazy_any:nTF
      {
```

```
        { \__text_codepoint_compare_p:nNn {#4} = { "0389 } }
        { \__text_codepoint_compare_p:nNn {#4} = { "03AE } }
        { \__text_codepoint_compare_p:nNn {#4} = { "1F22 } }
        { \__text_codepoint_compare_p:nNn {#4} = { "1F2A } }
      }
      { \__text_change_case_boundary_upper_el:nnnnw {#1} {#2} {#3} {#4} }
      { \__text_change_case_breathing:nnnn {#1} {#2} {#3} {#4} }
  }
\cs_new:Npn \__text_change_case_boundary_upper_el:nnnnw
  #1#2#3#4#5 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#5}
      { \__text_change_case_loop:nnnw {#1} {#2} {#3} #4 }
      {
        \__text_change_case_store:e
          {
            \codepoint_generate:nn { "0389 }
              { \__text_change_case_catcode:nn {#4} { "0389 } }
          }
        \__text_change_case_loop:nnnw {#1} {#2} {#3}
      }
        #5 \q__text_recursion_stop
  }
```

(*End of definition for* `\__text_change_case_boundary_upper_el:Nnnnw` *and others.*)

__text_change_case_breathing:nnnn
__text_change_case_breathing:nnnnn
__text_change_case_breathing:nnnnnw
__text_change_case_breathing:nnnnnnw
__text_change_case_breathing_aux:nnnnnn
__text_change_case_breathing_aux:nnnnw
__text_change_case_breathing_aux:nnnN
__text_change_case_breathing_dialytika:nnnn

In Greek, breathing diacritics are normally dropped when uppercasing: see the code for the general case. However, for the first character of a word, if there is a breather *and* the next character takes a *dialytika*, it needs to be added. We start by checking if the current codepoint is in the Greek range, then decomposing.

```
\cs_new:Npn \__text_change_case_breathing:nnnn #1#2#3#4
  {
    \__text_change_case_if_greek:nTF {#4}
      {
        \exp_args:Ne \__text_change_case_breathing:nnnnn
          {
            \codepoint_to_nfd:n
              { \__text_codepoint_from_chars:Nw #4 }
          }
            {#1} {#2} {#3} {#4}
      }
      { \__text_change_case_loop:nnnw {#1} {#2} {#3} #4 }
  }
\cs_new:Npn \__text_change_case_breathing:nnnnn #1#2#3#4#5
  {
    \__text_codepoint_process:nN
      { \__text_change_case_breathing:nnnnnw {#2} {#3} {#4} {#5} }
        #1 \q_mark
  }
```

Normal form decomposition will always give between one and three codepoints. Luckily, the two breathing marks (*psili* and *dasia*) will be in a predictable position: last. So we can quickly establish first that there was a change on decomposition, and second if the final resulting codepoint is one of the two we care about.

```
\cs_new:Npn \__text_change_case_breathing:nnnnnw #1#2#3#4#5#6 \q_mark
  {
    \tl_if_blank:nTF {#6}
      { \__text_change_case_loop:nnnw {#1} {#2} {#3} #4 }
      {
        \__text_codepoint_process:nN
          { \__text_change_case_breathing:nnnnnnw {#1} {#2} {#3} {#4} {#5} }
            #6 \q_mark
      }
  }
\cs_new:Npn \__text_change_case_breathing:nnnnnnw #1#2#3#4#5#6#7 \q_mark
  {
    \tl_if_blank:nTF {#7}
      {
        \__text_change_case_breathing_aux:nnnnnn
          {#1} {#2} {#3} {#4} {#5} {#6}
      }
      {
        \__text_codepoint_process:nN
          { \__text_change_case_breathing:nnnnnnw {#1} {#2} {#3} {#4} {#5} }
          #7 \q_mark
      }
  }
\cs_new:Npn \__text_change_case_breathing_aux:nnnnnn #1#2#3#4#5#6
  {
    \bool_lazy_or:nnTF
      { \__text_codepoint_compare_p:nNn {#6} = { "0313 } }
      { \__text_codepoint_compare_p:nNn {#6} = { "0314 } }
      { \__text_change_case_breathing_aux:nnnnw {#1} {#2} {#3} {#5} }
      { \__text_change_case_loop:nnnw {#1} {#2} {#3} #4 }
  }
```

Now the lookahead can be fired: check the next codepoint and assess whether it takes a *dialytika*. Drop the breathing mark or generate the *dialytika*: the latter is code shared with the general mechanism.

```
\cs_new:Npn \__text_change_case_breathing_aux:nnnnw #1#2#3#4#5
  \q__text_recursion_stop
  {
    \__text_change_case_store:e
      { \__text_change_case_codepoint:nn { upper } {#4} }
    \tl_if_head_is_N_type:nTF {#5}
      { \__text_change_case_breathing_aux:nnnN }
      { \__text_change_case_loop:nnnw }
        {#1} {#2} {#3} #5 \q__text_recursion_stop
  }
\cs_new:Npn \__text_change_case_breathing_aux:nnnN #1#2#3#4
  {
    \__text_codepoint_process:nN
      { \__text_change_case_breathing_dialytika:nnnn {#1} {#2} {#3} } #4
  }
\cs_new:Npn \__text_change_case_breathing_dialytika:nnnn #1#2#3#4
  {
    \__text_change_case_if_takes_dialytika:nTF {#4}
      {
```

```
      \__text_change_case_upper_el_dialytika:n {#4}
      \__text_change_case_loop:nnnw {#1} {#2} {#3}
    }
    { \__text_change_case_loop:nnnw {#1} {#2} {#3} #4 }
}
```

(*End of definition for* `\__text_change_case_breathing:nnnn` *and others.*)

`\__text_change_case_title_el:nnnnn`

Titlecasing retains accents, but to prevent the uppercasing code from kicking in, there has to be an explicit function here.

```
\cs_new:Npn \__text_change_case_title_el:nnnnn #1#2#3#4#5
  { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
```

(*End of definition for* `\__text_change_case_title_el:nnnnn`.)

`\__text_change_case_upper_hy:nnnnn`
`\__text_change_case_title_hy:nnnnn`
`\__text_change_case_upper_hy-x-yiwn:nnnnn`
`\__text_change_case_title_hy-x-yiwn:nnnnn`

See https://www.unicode.org/L2/L2020/20143-armenian-ech-yiwn.pdf.

```
\cs_new:Npn \__text_change_case_upper_hy:nnnnn #1#2#3#4#5
  {
    \__text_codepoint_compare:nNnTF {#5} = { "0587 }
      {
        \__text_change_case_store:e
          {
            \codepoint_generate:nn { "0535 }
              { \__text_change_case_catcode:nn {#5} { "0535 } }
            \codepoint_generate:nn { "054E }
              { \__text_change_case_catcode:nn {#5} { "054E } }
          }
        \use:c { __text_change_case_next_ #2 :nnn }
          {#2} {#3} {#4}
      }
      { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
  }
\cs_new:Npn \__text_change_case_title_hy:nnnnn #1#2#3#4#5
  {
    \__text_codepoint_compare:nNnTF {#5} = { "0587 }
      {
        \__text_change_case_store:e
          {
            \codepoint_generate:nn { "0535 }
              { \__text_change_case_catcode:nn {#5} { "0535 } }
            \codepoint_generate:nn { "057E }
              { \__text_change_case_catcode:nn {#5} { "057E } }
          }
        \use:c { __text_change_case_next_ #2 :nnn }
          {#2} {#3} {#4}
      }
      { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
  }
\cs_new:cpn { __text_change_case_upper_hy-x-yiwn:nnnnn } #1#2#3#4#5
  { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
\cs_new_eq:cc { __text_change_case_title_hy-x-yiwn:nnnnn }
  { __text_change_case_upper_hy-x-yiwn:nnnnn }
```

(*End of definition for* `\__text_change_case_upper_hy:nnnnn` *and others.*)

_text_change_case_lower_la-x-medieval:nnnnn
_text_change_case_upper_la-x-medieval:nnnnn

Simply swaps of characters.

```
\cs_new:cpn { __text_change_case_lower_la-x-medieval:nnnnn } #1#2#3#4#5
  {
    \__text_codepoint_compare:nNnTF {#5} = { "0056 }
      {
        \__text_change_case_store:e
          { \char_generate:nn { "0075 } { \__text_char_catcode:N #5 } }
        \use:c { __text_change_case_next_ #2 :nnn }
          {#2} {#3} {#4}
      }
      { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
  }
\cs_new:cpn { __text_change_case_upper_la-x-medieval:nnnnn } #1#2#3#4#5
  {
    \__text_codepoint_compare:nNnTF {#5} = { "0075 }
      {
        \__text_change_case_store:e
          { \char_generate:nn { "0056 } { \__text_char_catcode:N #5 } }
        \use:c { __text_change_case_next_ #2 :nnn }
          {#2} {#3} {#4}
      }
      { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
  }
```

(*End of definition for* `\__text_change_case_lower_la-x-medieval:nnnnn` *and* `\__text_change_case_upper_la-x-medieval:nnnnn`.)

__text_change_cases_lower_lt:nnnnn
__text_change_cases_lower_lt_auxi:nnnnn
__text_change_cases_lower_lt_auxii:nnnnn
__text_change_case_lower_lt:nnnw
__text_change_case_lower_lt:nnnN
__text_change_case_lower_lt:nnnn

For Lithuanian, the issue to be dealt with is dots over lower case letters: these should be present if there is another accent. The first step is a simple match attempt: look for the three uppercase accented letters which should gain a dot-above char in their lowercase form.

```
\cs_new:Npn \__text_change_case_lower_lt:nnnnn #1#2#3#4#5
  {
    \exp_args:Ne \__text_change_case_lower_lt_auxi:nnnnn
      {
        \int_case:nn { \__text_codepoint_from_chars:Nw #5 }
          {
            { "00CC } { "0300 }
            { "00CD } { "0301 }
            { "0128 } { "0303 }
          }
      }
        {#2} {#3} {#4} {#5}
  }
```

If there was a hit, output the result with the dot-above and move on. Otherwise, look for one of the three letters that can take a combining accent: I, J, and I-ogonek.

```
\cs_new:Npn \__text_change_case_lower_lt_auxi:nnnnn #1#2#3#4#5
  {
    \tl_if_blank:nTF {#1}
      {
        \exp_args:Ne \__text_change_case_lower_lt_auxii:nnnnn
          {
            \int_case:nn { \__text_codepoint_from_chars:Nw #5 }
```

```
            {
              { "0049 } { "0069 }
              { "004A } { "006A }
              { "012E } { "012F }
            }
        }
          {#2} {#3} {#4} {#5}
    }
    {
      \__text_change_case_store:e
        {
          \codepoint_generate:nn { "0069 }
            { \__text_change_case_catcode:nn {#5} { "0069 } }
          \codepoint_generate:nn { "0307 }
            { \__text_change_case_catcode:nn {#5} { "0307 } }
          \codepoint_generate:nn {#1}
            { \__text_change_case_catcode:nn {#5} {#1} }
        }
      \__text_change_case_loop:nnnw {#2} {#3} {#4}
    }
}
```

Again, branch depending on a hit. If there is one, we output the character then need to look for a combining accent: as usual, we need to be aware of the loop situation.

```
\cs_new:Npn \__text_change_case_lower_lt_auxii:nnnnn #1#2#3#4#5
  {
    \tl_if_blank:nTF {#1}
      { \__text_change_case_codepoint:nnnnn {#2} {#2} {#3} {#4} {#5} }
      {
        \__text_change_case_store:e
          {
            \codepoint_generate:nn {#1}
              { \__text_change_case_catcode:nn {#5} {#1} }
          }
        \__text_change_case_lower_lt:nnnw {#2} {#3} {#4}
      }
  }
\cs_new:Npn \__text_change_case_lower_lt:nnnw #1#2#3#4 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#4}
      { \__text_change_case_lower_lt:nnnN }
      { \__text_change_case_loop:nnnw }
        {#1} {#2} {#3} #4 \q__text_recursion_stop
  }
\cs_new:Npn \__text_change_case_lower_lt:nnnN #1#2#3#4
  {
    \__text_codepoint_process:nN
      { \__text_change_case_lower_lt:nnnn {#1} {#2} {#3} } #4
  }
\cs_new:Npn \__text_change_case_lower_lt:nnnn #1#2#3#4
  {
    \bool_lazy_and:nnT
      {
        \bool_lazy_or_p:nn
```

```
          { ! \tl_if_single_p:n {#4} }
          { ! \token_if_cs_p:N #4 }
      }
      {
        \bool_lazy_any_p:n
          {
            { \__text_codepoint_compare_p:nNn {#4} = { "0300 } }
            { \__text_codepoint_compare_p:nNn {#4} = { "0301 } }
            { \__text_codepoint_compare_p:nNn {#4} = { "0303 } }
          }
      }
      {
        \__text_change_case_store:e
          {
            \codepoint_generate:nn { "0307 }
              { \__text_change_case_catcode:nn {#4} { "0307 } }
          }
      }
    \__text_change_case_loop:nnnw {#1} {#2} {#3} #4
  }
```

(*End of definition for* `\__text_change_cases_lower_lt:nnnnn` *and others.*)

`\__text_change_cases_upper_lt:nnnnn`
`\__text_change_cases_upper_lt_aux:nnnnn`
`\__text_change_case_upper_lt:nnnw`
`\__text_change_case_upper_lt:nnnN`
`\__text_change_case_upper_lt:nnnn`

The uppercasing version: first find i/j/i-ogonek, then look for the combining char: drop it if present.

```
\cs_new:Npn \__text_change_case_upper_lt:nnnnn #1#2#3#4#5
  {
    \exp_args:Ne \__text_change_case_upper_lt_aux:nnnnn
      {
        \int_case:nn { \__text_codepoint_from_chars:Nw #5 }
          {
            { "0069 } { "0049 }
            { "006A } { "004A }
            { "012F } { "012E }
          }
      }
      {#2} {#3} {#4} {#5}
  }
\cs_new:Npn \__text_change_case_upper_lt_aux:nnnnn #1#2#3#4#5
  {
    \tl_if_blank:nTF {#1}
      { \__text_change_case_codepoint:nnnnn { upper } {#2} {#3} {#4} {#5} }
      {
        \__text_change_case_store:e
          {
            \codepoint_generate:nn {#1}
              { \__text_change_case_catcode:nn {#5} {#1} }
          }
        \__text_change_case_upper_lt:nnnw {#2} {#3} {#4}
      }
  }
\cs_new:Npn \__text_change_case_upper_lt:nnnw #1#2#3#4 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#4}
```

```
      { \__text_change_case_upper_lt:nnnN }
      { \use:c { __text_change_case_next_ #1 :nnn } }
        {#1} {#2} {#3} #4 \q__text_recursion_stop
  }
\cs_new:Npn \__text_change_case_upper_lt:nnnN #1#2#3#4
  {
    \__text_codepoint_process:nN
      { \__text_change_case_upper_lt:nnnn {#1} {#2} {#3} } #4
  }
\cs_new:Npn \__text_change_case_upper_lt:nnnn #1#2#3#4
  {
    \bool_lazy_and:nnTF
      {
        \bool_lazy_or_p:nn
          { ! \tl_if_single_p:n {#4} }
          { ! \token_if_cs_p:N #4 }
      }
      { \__text_codepoint_compare_p:nNn {#4} = { "0307 } }
      { \use:c { __text_change_case_next_ #1 :nnn } {#1} {#2} {#3} }
      { \use:c { __text_change_case_next_ #1 :nnn } {#1} {#2} {#3} #4 }
  }
```

(*End of definition for* `\__text_change_cases_upper_lt:nnnnn` *and others.*)

__text_change_case_title_nl:nnnnn
__text_change_case_title_nl_aux:nnnnn
__text_change_case_title_nl:nnnw
__text_change_case_title_nl:nnnN

For Dutch, there is a single look-ahead test for `ij` when title casing. If the appropriate letters are found, produce `IJ` and gobble the `j`/`J`.

```
\cs_new:Npn \__text_change_case_title_nl:nnnnn #1#2#3#4#5
  {
    \tl_if_single:nTF {#5}
      { \__text_change_case_title_nl_aux:nnnnn }
      { \__text_change_case_codepoint:nnnnn }
        {#1} {#2} {#3} {#4}  {#5}
  }
\cs_new:Npn \__text_change_case_title_nl_aux:nnnnn #1#2#3#4#5
  {
    \bool_lazy_or:nnTF
      { \int_compare_p:nNn {`#5} = { "0049 } }
      { \int_compare_p:nNn {`#5} = { "0069 } }
      {
        \__text_change_case_store:e
          { \char_generate:nn { "0049 } { \__text_char_catcode:N #5 } }
        \__text_change_case_title_nl:nnnw {#2} {#3} {#4}
      }
      { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
  }
\cs_new:Npn \__text_change_case_title_nl:nnnw #1#2#3#4 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#4}
      { \__text_change_case_title_nl:nnnN }
      { \use:c { __text_change_case_next_ #1 :nnn } }
        {#1} {#2} {#3} #4 \q__text_recursion_stop
  }
\cs_new:Npn \__text_change_case_title_nl:nnnN #1#2#3#4
  {
```

```
    \bool_lazy_and:nnTF
      { ! \token_if_cs_p:N #4 }
      {
        \bool_lazy_or_p:nn
          { \int_compare_p:nNn {‘#4} = { "004A } }
          { \int_compare_p:nNn {‘#4} = { "006A } }
      }
      {
        \__text_change_case_store:e
          { \char_generate:nn { "004A } { \__text_char_catcode:N #4 } }
        \use:c { __text_change_case_next_ #1 :nnn } {#1} {#2} {#3}
      }
      { \use:c { __text_change_case_next_ #1 :nnn } {#1} {#2} {#3} #4 }
  }
```

(*End of definition for* `\__text_change_case_title_nl:nnnnn` *and others.*)

`\__text_change_case_lower_tr:nnnnn`
`\__text_change_case_lower_tr:nnnNw`
`\__text_change_case_lower_tr:NnnnN`
`\__text_change_case_lower_tr:Nnnnn`

The Turkic languages need special treatment for dotted-i and dotless-i. The lower casing rule can be expressed in terms of searching first for either a dotless-I or a dotted-I. In the latter case the mapping is easy, but in the former there is a second stage search.

```
\cs_new:Npn \__text_change_case_lower_tr:nnnnn #1#2#3#4#5
  {
    \__text_codepoint_compare:nNnTF {#5} = { "0049 }
      { \__text_change_case_lower_tr:nnnNw {#1} {#3} {#4} #5 }
      {
        \__text_codepoint_compare:nNnTF {#5} = { "0130 }
          {
            \__text_change_case_store:e
              {
                \codepoint_generate:nn { "0069 }
                  { \__text_change_case_catcode:nn {#5} { "0069 } }
              }
            \__text_change_case_loop:nnnw {#1} {#3} {#4}
          }
          { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
      }
  }
```

After a dotless-I there may be a dot-above character. If there is then a dotted-i should be produced, otherwise output a dotless-i. When the combination is found both the dotless-I and the dot-above char have to be removed from the input.

```
\cs_new:Npn \__text_change_case_lower_tr:nnnNw #1#2#3#4#5 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#5}
      { \__text_change_case_lower_tr:NnnnN  #4 {#1} {#2} {#3} }
      {
        \__text_change_case_store:e
          {
            \codepoint_generate:nn { "0131 }
              { \__text_change_case_catcode:nn {#4} { "0131 } }
          }
        \__text_change_case_loop:nnnw {#1} {#2} {#3}
      }
        #5 \q__text_recursion_stop
```

```
  }
\cs_new:Npn \__text_change_case_lower_tr:NnnnN #1#2#3#4#5
  {
    \__text_codepoint_process:nN
      { \__text_change_case_lower_tr:Nnnnn #1 {#2} {#3} {#4} } #5
  }
\cs_new:Npn \__text_change_case_lower_tr:Nnnnn #1#2#3#4#5
  {
    \bool_lazy_or:nnTF
      {
        \bool_lazy_and_p:nn
          { \tl_if_single_p:n {#5} }
          { \token_if_cs_p:N #5 }
      }
      { ! \__text_codepoint_compare_p:nNn {#5} = { "0307 } }
      {
        \__text_change_case_store:e
          {
            \codepoint_generate:nn { "0131 }
              { \__text_change_case_catcode:nn {#1} { "0131 } }
          }
        \__text_change_case_loop:nnnw {#2} {#3} {#4} #5
      }
      {
        \__text_change_case_store:e
          {
            \codepoint_generate:nn { "0069 }
              { \__text_change_case_catcode:nn {#1} { "0069 } }
          }
        \__text_change_case_loop:nnnw {#2} {#3} {#4}
      }
  }
```

(*End of definition for* `\__text_change_case_lower_tr:nnnnn` *and others.*)

`\__text_change_case_upper_tr:nnnnn`

Uppercasing is easier: just one exception with no context.

```
\cs_new:Npn \__text_change_case_upper_tr:nnnnn #1#2#3#4#5
  {
    \__text_codepoint_compare:nNnTF {#5} = { "0069 }
      {
        \__text_change_case_store:e
          {
            \codepoint_generate:nn { "0130 }
              { \__text_change_case_catcode:nn {#5} { "0130 } }
          }
        \use:c { __text_change_case_next_ #2 :nnn } {#2} {#3} {#4}
      }
      { \__text_change_case_codepoint:nnnnn {#1} {#2} {#3} {#4} {#5} }
  }
```

(*End of definition for* `\__text_change_case_upper_tr:nnnnn`*.*)

`\__text_change_case_lower_az:nnnnn`
`\__text_change_case_upper_az:nnnnn`

Straight copies.

```
\cs_new_eq:NN \__text_change_case_lower_az:nnnnn
  \__text_change_case_lower_tr:nnnnn
```

```
\cs_new_eq:NN \__text_change_case_upper_az:nnnnn
  \__text_change_case_upper_tr:nnnnn
```

(*End of definition for* `\__text_change_case_lower_az:nnnnn` *and* `\__text_change_case_upper_az:nnnnn`*.*)

The (fixed) look-up mappings for letter-like control sequences.

```
\group_begin:
  \cs_set_protected:Npn \__text_change_case_setup:NN #1#2
    {
      \quark_if_recursion_tail_stop:N #1
      \tl_const:cn { c__text_lowercase_ \token_to_str:N #1 _tl }
        { #2 }
      \tl_const:cn { c__text_uppercase_ \token_to_str:N #2 _tl }
        { #1 }
      \__text_change_case_setup:NN
    }
  \__text_change_case_setup:NN
  \AA \aa
  \AE \ae
  \DH \dh
  \DJ \dj
  \IJ \ij
  \L  \l
  \NG \ng
  \O  \o
  \OE \oe
  \SS \ss
  \TH \th
  \q_recursion_tail ?
  \q_recursion_stop
  \tl_const:cn { c__text_uppercase_ \token_to_str:N \i _tl } { I }
  \tl_const:cn { c__text_uppercase_ \token_to_str:N \j _tl } { J }
\group_end:
```

To deal with possible encoding-specific extensions to `\@uclclist`, we check at the end of the preamble. This will therefore only apply to LaTeX 2_ε package mode.

```
\tl_if_exist:NT \@expl@finalise@setup@@
  {
    \tl_gput_right:Nn \@expl@finalise@setup@@
      {
        \tl_gput_right:Nn \@kernel@after@begindocument
          {
            \group_begin:
              \cs_set_protected:Npn \__text_change_case_setup:Nn #1#2
                {
                  \quark_if_recursion_tail_stop:N #1
                  \tl_if_single_token:nT {#2}
                    {
                      \cs_if_exist:cF
                        { c__text_uppercase_ \token_to_str:N #1 _tl }
                        {
                          \tl_const:cn
                            { c__text_uppercase_ \token_to_str:N #1 _tl }
                            { #2 }
                        }
```

```
           \cs_if_exist:cF
            { c__text_lowercase_ \token_to_str:N #2 _tl }
            {
             \tl_const:cn
              { c__text_lowercase_ \token_to_str:N #2 _tl }
              { #1 }
            }
          }
         \__text_change_case_setup:Nn
        }
       \exp_after:wN \__text_change_case_setup:Nn \@uclclist
       \q_recursion_tail ?
       \q_recursion_stop
      \group_end:
     }
   }
 }
```

A few adjustments to case mapping for combining chars: these are not needed for the Unicode engines

```
\bool_lazy_or:nnF
 { \sys_if_engine_luatex_p: }
 { \sys_if_engine_xetex_p: }
 {
   \text_declare_uppercase_mapping:nn { "01F0 } { \v { J } }
 }
⟨/package⟩
```

Chapter 89

l3text-map implementation

```
⟨*package⟩
⟨@@=text⟩
```

89.1 Mapping to text

\text_map_function:nN
__text_map_function:nN
__text_map_loop:Nnw
__text_map_group:Nnn
__text_map_space:Nnw
__text_map_N_type:NnN
__text_map_codepoint:Nnn
__text_map_CR:Nnw
__text_map_CR:NnN
__text_map_class:Nnnn
__text_map_class:nNnnn
__text_map_class_loop:Nnnnw
__text_map_class_end:nw
__text_map_Control:Nnn
__text_map_Extend:Nnn
__text_map_SpacingMark:Nnn
__text_map_Prepend:Nnn
__text_map_Prepend_aux:Nnn
__text_map_Prepend:nNnn
__text_map_Prepend_loop:Nnnw
__text_map_not_Control:Nnn
__text_map_not_Extend:Nnn
__text_map_not_SpacingMark:Nnn
__text_map_not_Prepend:Nnn
__text_map_not_L:Nnn
__text_map_not_LV:Nnn
__text_map_not_V:Nnn
__text_map_not_LVT:Nnn
__text_map_not_T:Nnn
__text_map_L:Nnn
__text_map_LV:Nnn
__text_map_V:Nnn
__text_map_LVT:Nnn
__text_map_T:Nnn
__text_map_hangul:Nnnw
__text_map_hangul:NnnN
__text_map_hangul:Nnnn
__text_map_hangul_aux:Nnnnw
__text_map_hangul:nNnnnw
__text_map_hangul_loop:Nnnnnw

The standard lead-off for an action loop.

```
\cs_new:Npn \text_map_function:nN #1#2
  { \exp_args:Ne \__text_map_function:nN { \text_expand:n {#1} } #2 }
\cs_new:Npn \__text_map_function:nN #1#2
  {
    \__text_map_loop:Nnw #2 { } #1
      \q__text_recursion_tail \q__text_recursion_stop
    \prg_break_point:Nn \text_map_break: { }
  }
```

The standard set up for an "action" loop. Groups are handled by recursion, spaces are treated similarly: both count as grapheme boundaries. For N-type tokens, we filter out control sequences (again a boundary), then move on to further analysis.

```
\cs_new:Npn \__text_map_loop:Nnw #1#2#3 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#3}
      { \__text_map_N_type:NnN }
      {
        \tl_if_head_is_group:nTF {#3}
          { \__text_map_group:Nnn }
          { \__text_map_space:Nnw }
      }
    #1 {#2} #3 \q__text_recursion_stop
  }
\cs_new:Npn \__text_map_group:Nnn #1#2#3
  {
    \__text_map_output:Nn #1 {#2}
    {
      \__text_map_loop:Nnw #1 { } #2
        \q__text_recursion_tail \q__text_recursion_stop
      \prg_break_point:Nn \text_map_break: { }
    }
```

```
    \__text_map_loop:Nnw #1 { }
  }
\use:e
  { \cs_new:Npn \exp_not:N \__text_map_space:Nnw #1#2 \c_space_tl }
  {
    \__text_map_output:Nn #1 {#2}
    #1 { ~ }
    \__text_map_loop:Nnw #1 { }
  }
\cs_new:Npn \__text_map_N_type:NnN #1#2#3
  {
    \__text_if_q_recursion_tail_stop_do:Nn #3
      {
        \__text_map_output:Nn #1 {#2}
        \text_map_break:
      }
    \token_if_cs:NTF #3
      {
        \__text_map_output:Nn #1 {#2}
        #1 {#3}
        \__text_map_loop:Nnw #1 { }
      }
      {
        \__text_codepoint_process:nN
          { \__text_map_codepoint:Nnn #1 {#2} } #3
      }
  }
```

We pull out a few special cases here. Carriage returns case needs a bit of context handling so has an auxiliary. Codepoint U+200D is the zero-width joiner, which has no context to concern us: just don't break.

```
\cs_new:Npn \__text_map_codepoint:Nnn #1#2#3
  {
    \__text_codepoint_compare:nNnTF {#3} = { "0D }
      {
        \__text_map_output:Nn #1 {#2}
        \__text_map_CR:Nnw #1 {#3}
      }
      {
        \__text_codepoint_compare:nNnTF {#3} = { "200D }
          { \__text_map_loop:Nnw #1 {#2#3} }
          { \__text_map_class:Nnnn #1 {#2} {#3} { Control } }
      }
  }
```

A carriage return is a boundary unless it is immediately followed by a line feed, in which case that pair is a boundary.

```
\cs_new:Npn \__text_map_CR:Nnw #1#2#3 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#3}
      { \__text_map_CR:NnN #1 {#2} }
      {
        #1 {#2}
        \__text_map_loop:Nnw #1 { }
      }
```

```
      #3 \q__text_recursion_stop
  }
\cs_new:Npn \__text_map_CR:NnN #1#2#3
  {
    \__text_if_q_recursion_tail_stop_do:Nn #3
      {
        #1 {#2}
        \text_map_break:
      }
    \bool_lazy_and:nnTF
      { ! \token_if_cs_p:N #3 }
      { \int_compare_p:nNn { `#3 } = { "0A } }
      {
        \__text_map_output:Nn #1 {#2#3}
        \__text_map_loop:Nnw #1 { }
      }
      { \__text_map_loop:Nnw #1 { } #3 }
  }
```

There are various classes of character, and we deal with them all in the same general way. We need to example the relevant list of codepoints: if we get a hit, then we do whatever the relevant action is. Otherwise we loop, but only if the current codepoint could still match: the loop stops early otherwise and we move forward.

```
\cs_new:Npn \__text_map_class:Nnnn #1#2#3#4
  {
    \exp_args:Nv \__text_map_class:nNnnn { c__text_grapheme_ #4 _clist }
      #1 {#2} {#3} {#4}
  }
\cs_new:Npn \__text_map_class:nNnnn #1#2#3#4#5
  {
    \__text_map_class_loop:Nnnnw #2 {#3} {#4} {#5}
      #1 , \q__text_recursion_tail .. , \q__text_recursion_stop
  }
\cs_new:Npn \__text_map_class_loop:Nnnnw #1#2#3#4 #5 .. #6 ,
  {
    \__text_if_q_recursion_tail_stop_do:nn {#5}
      { \use:c { __text_map_not_ #4 :Nnn } #1 {#2} {#3} }
    \__text_codepoint_compare:nNnTF {#3} < { "#5 }
      {
        \__text_map_class_end:nw
          { \use:c { __text_map_not_ #4 :Nnn } #1 {#2} {#3} }
      }
      {
        \__text_codepoint_compare:nNnTF {#3} > { "#6 }
          { \__text_map_class_loop:Nnnnw #1 {#2} {#3} {#4} }
          {
            \__text_map_class_end:nw
              { \use:c { __text_map_ #4 :Nnn } #1 {#2} {#3} }
          }
      }
  }
\cs_new:Npn \__text_map_class_end:nw #1#2 \q__text_recursion_stop {#1}
```

Break before *and* after.

```
\cs_new:Npn \__text_map_Control:Nnn #1#2#3
```

```
  {
    \__text_map_output:Nn #1 {#2}
    \__text_map_output:Nn #1 {#3}
    \__text_map_loop:Nnw #1 { }
  }
```

Keep collecting.

```
\cs_new:Npn \__text_map_Extend:Nnn #1#2#3
  { \__text_map_loop:Nnw #1 {#2#3} }
\cs_new_eq:NN \__text_map_SpacingMark:Nnn \__text_map_Extend:Nnn
```

Outputting anything earlier, the combine with what follows. The only exclusions are control characters.

```
\cs_new:Npn \__text_map_Prepend:Nnn #1#2#3
  {
    \__text_map_output:Nn #1 {#2}
    \__text_map_lookahead:NnNw #1 {#3} \__text_map_Prepend_aux:Nnn
  }
\cs_new:Npn \__text_map_Prepend_aux:Nnn #1#2#3
  {
    \bool_lazy_or:nnTF
      { \__text_codepoint_compare_p:nNn {#3} = { "0A } }
      { \__text_codepoint_compare_p:nNn {#3} = { "0D } }
      {
        #1 {#2}
        \__text_map_loop:Nnw #1 {#3}
      }
      {
        \exp_args:NV \__text_map_Prepend:nNnn
          \c__text_grapheme_Control_clist
          #1 {#2} {#3}
      }
  }
\cs_new:Npn \__text_map_Prepend:nNnn #1#2#3#4
  {
    \__text_map_Prepend_loop:Nnnw #2 {#3} {#4}
      #1 , \q__text_recursion_tail .. , \q__text_recursion_stop
  }
\cs_new:Npn \__text_map_Prepend_loop:Nnnw #1#2#3 #4 .. #5 ,
  {
    \__text_if_q_recursion_tail_stop_do:nn {#4}
      { \__text_map_loop:Nnw #1 {#2#3} }
    \__text_codepoint_compare:nNnTF {#3} < { "#4 }
      {
        \__text_map_class_end:nw
          { \__text_map_loop:Nnw #1 {#2#3} }
      }
      {
        \__text_codepoint_compare:nNnTF {#3} > { "#5 }
          { \__text_map_Prepend_loop:Nnnw #1 {#2} {#3} }
          {
            \__text_map_class_end:nw
              { \__text_map_loop:Nnw #1 {#2} #3 }
          }
      }
```

```
  }
```

Dealing with end-of-class is done such that we can be flexible.

```
\cs_new:Npn \__text_map_not_Control:Nnn #1#2#3
  { \__text_map_class:Nnnn #1 {#2} {#3} { Extend } }
\cs_new:Npn \__text_map_not_Extend:Nnn #1#2#3
  { \__text_map_class:Nnnn #1 {#2} {#3} { SpacingMark } }
\cs_new:Npn \__text_map_not_SpacingMark:Nnn #1#2#3
  { \__text_map_class:Nnnn #1 {#2} {#3} { Prepend } }
\cs_new:Npn \__text_map_not_Prepend:Nnn #1#2#3
  { \__text_map_class:Nnnn #1 {#2} {#3} { L } }
\cs_new:Npn \__text_map_not_L:Nnn #1#2#3
  { \__text_map_class:Nnnn #1 {#2} {#3} { LV } }
\cs_new:Npn \__text_map_not_LV:Nnn #1#2#3
  { \__text_map_class:Nnnn #1 {#2} {#3} { V } }
\cs_new:Npn \__text_map_not_V:Nnn #1#2#3
  { \__text_map_class:Nnnn #1 {#2} {#3} { LVT } }
\cs_new:Npn \__text_map_not_LVT:Nnn #1#2#3
  { \__text_map_class:Nnnn #1 {#2} {#3} { T } }
\cs_new:Npn \__text_map_not_T:Nnn #1#2#3
  { \__text_map_class:Nnnn #1 {#2} {#3} { Regional_Indicator } }
\cs_new:Npn \__text_map_not_Regional_Indicator:Nnn #1#2#3
  {
    \__text_map_output:Nn #1 {#2}
    \__text_map_loop:Nnw #1 {#3}
  }
```

Hangul needs additional treatment. First we have to deal with the start-of-Hangul position: output what we had up to now, then move the specialist handler. The idea here is to pick off the different codepoint types one at a time, tracking what else can be considered at each stage until we hit the end of the viable types. Other than that, we just keep building up the Hangul codepoints using a dedicated version of the loop from above.

```
\cs_new:Npn \__text_map_L:Nnn #1#2#3
  {
    \__text_map_output:Nn #1 {#2}
    \__text_map_hangul:Nnnw
      #1 {#3} { L ; V ; LV ; LVT }
  }
\cs_new:Npn \__text_map_LV:Nnn #1#2#3
  {
    \__text_map_output:Nn #1 {#2}
    \__text_map_hangul:Nnnw
      #1 {#3} { V ; T }
  }
\cs_new_eq:NN \__text_map_V:Nnn \__text_map_LV:Nnn
\cs_new:Npn \__text_map_LVT:Nnn #1#2#3
  {
    \__text_map_output:Nn #1 {#2}
    \__text_map_hangul:Nnnw
      #1 {#3} { T }
  }
\cs_new_eq:NN \__text_map_T:Nnn \__text_map_LVT:Nnn
\cs_new:Npn \__text_map_hangul:Nnnw #1#2#3#4 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#4}
```

```
      { \__text_map_hangul:NnnN #1 {#2} {#3} }
      {
        #1 {#2}
        \__text_map_loop:Nnw #1 { }
      }
    #4 \q__text_recursion_stop
  }
\cs_new:Npn \__text_map_hangul:NnnN #1#2#3#4
  {
    \__text_if_q_recursion_tail_stop_do:Nn #4
      {
        #1 {#2}
        \text_map_break:
      }
    \token_if_cs:NTF #4
      {
        #1 {#2}
        \__text_map_loop:Nnw #1 { }
      }
      {
        \__text_codepoint_process:nN
          { \__text_map_hangul:Nnnn #1 {#2} {#3} } #4
      }
  }
\cs_new:Npn \__text_map_hangul:Nnnn #1#2#3#4
  {
    \__text_map_hangul_aux:Nnnw #1 {#2} {#4}
      #3 ; \q_recursion_tail ; \q_recursion_stop
  }
\cs_new:Npn \__text_map_hangul_aux:Nnnw #1#2#3#4 ;
  {
    \quark_if_recursion_tail_stop_do:nn {#4}
      { \__text_map_loop:Nnw #1 {#2} #3 }
    \exp_args:Nv \__text_map_hangul:nNnnnw { c__text_grapheme_ #4 _clist }
      #1 {#2} {#3} {#4}
  }
\cs_new:Npn \__text_map_hangul:nNnnnw #1#2#3#4#5#6  \q_recursion_stop
  {
    \__text_map_hangul_loop:Nnnnnw #2 {#3} {#4} {#5} {#6}
      #1 , \q__text_recursion_tail .. , \q__text_recursion_stop
  }
\cs_new:Npn \__text_map_hangul_loop:Nnnnnw #1#2#3#4#5 #6 .. #7 ,
  {
    \__text_if_q_recursion_tail_stop_do:nn {#6}
      { \__text_map_hangul_next:Nnnn #1 {#2} {#3} {#5} }
    \__text_codepoint_compare:nNnTF {#3} < { "#6 }
      {
        \__text_map_hangul_end:nw
          { \__text_map_hangul_next:Nnnn #1 {#2} {#3} {#5} }
      }
      {
        \__text_codepoint_compare:nNnTF {#3} > { "#7 }
          { \__text_map_hangul_loop:Nnnnnw #1 {#2} {#3} {#4} {#5} }
          {
```

```
            \__text_map_hangul_end:nw
              { \use:c { __text_map_hangul_ #4 :Nnn } #1 {#2} {#3} }
          }
      }
  }
\cs_new:Npn \__text_map_hangul_next:Nnnn #1#2#3#4
  { \__text_map_hangul_aux:Nnnw #1 {#2} {#3} #4 \q_recursion_stop }
\cs_new:Npn \__text_map_hangul_end:nw #1#2 \q__text_recursion_stop {#1}
\cs_new:Npn \__text_map_hangul_L:Nnn #1#2#3
  {
    \__text_map_hangul:Nnnw
      #1 {#2#3} { L V { LV } { LVT } }
  }
\cs_new:Npn \__text_map_hangul_LV:Nnn #1#2#3
  {
    \__text_map_hangul:Nnnw
      #1 {#2#3} { VT }
  }
\cs_new_eq:NN \__text_map_hangul_V:Nnn \__text_map_hangul_LV:Nnn
\cs_new:Npn \__text_map_hangul_LVT:Nnn #1#2#3
  {
    \__text_map_hangul:Nnnw
      #1 {#2#3} { T }
  }
\cs_new_eq:NN \__text_map_hangul_T:Nnn \__text_map_hangul_LVT:Nnn
```

The Regional Indicator rule means looking ahead and dealing with the case where there are two in a row. So we use a look ahead to pick them off. As there is only one range the values are hard-coded.

```
\cs_new:Npn \__text_map_Regional_Indicator:Nnn #1#2#3
  {
    \__text_map_output:Nn #1 {#2}
    \__text_map_lookahead:NnNw #1 {#3} \__text_map_Regional_Indicator_aux:Nnn
  }
\cs_new:Npn \__text_map_Regional_Indicator_aux:Nnn #1#2#3
  {
    \bool_lazy_or:nnTF
      { \__text_codepoint_compare_p:nNn {#3} < { "1F1E6 } }
      { \__text_codepoint_compare_p:nNn {#3} > { "1F1FF } }
      {
        \__text_map_loop:Nnw #1 {#2} #3
      }
      { \__text_map_loop:Nnw #1 {#2#3} }
  }
```

A generic loop-ahead setup.

```
\cs_new:Npn \__text_map_lookahead:NnNw #1#2#3#4 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#4}
      { \__text_map_lookahead:NnNN #1 {#2} #3 }
      { \__text_map_loop:Nnw #1 {#2} }
    #4 \q__text_recursion_stop
  }
\cs_new:Npn \__text_map_lookahead:NnNN #1#2#3#4
  {
```

```
    \__text_if_q_recursion_tail_stop_do:Nn #4 { #1 {#2} }
    \token_if_cs:NTF #4
      {
        #1 {#2}
        \__text_map_loop:Nnw #1 { }
      }
      { \__text_codepoint_process:nN { #3 #1 {#2} } }
        #4
  }
```

For the end of the process.

```
\cs_new:Npn \__text_map_output:Nn #1#2
  { \tl_if_blank:nF {#2} { #1 {#2} } }
\cs_new:Npn \text_map_break:
  { \prg_map_break:Nn \text_map_break: { } }
\cs_new:Npn \text_map_break:n
  { \prg_map_break:Nn \text_map_break: }
```

(*End of definition for* `\text_map_function:nN` *and others. These functions are documented on page* *299*.)

`\text_map_inline:nn` The standard non-expandable inline version.

```
\cs_new_protected:Npn \text_map_inline:nn #1#2
  {
    \int_gincr:N \g__kernel_prg_map_int
    \cs_gset_protected:cpn
      { __text_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
    \exp_args:Nnc \text_map_function:nN {#1}
      { __text_map_ \int_use:N \g__kernel_prg_map_int :w }
    \prg_break_point:Nn \text_map_break:
      { \int_gdecr:N \g__kernel_prg_map_int }
  }
```

(*End of definition for* `\text_map_inline:nn`. *This function is documented on page* *299*.)

```
⟨/package⟩
```

Chapter 90

l3text-purify implementation

```
⟨*package⟩
⟨@@=text⟩
```

90.1 Purifying text

__text_if_recursion_tail_stop:N

Functions to query recursion quarks.

```
\__kernel_quark_new_test:N \__text_if_recursion_tail_stop:N
```

(*End of definition for* `\__text_if_recursion_tail_stop:N`.)

\text_purify:n
__text_purify:n
__text_purify_store:n
__text_purify_store:nw
__text_purify_end:w
__text_purify_loop:w
__text_purify_group:n
__text_purify_space:w
__text_purify_N_type:N
__text_purify_N_type_aux:N
__text_purify_math_search:NNN
__text_purify_math_start:NNw
__text_purify_math_store:n
__text_purify_math_store:nw
__text_purify_math_end:w
__text_purify_math_loop:NNw
__text_purify_math_N_type:NNN
__text_purify_math_group:NNn
__text_purify_math_space:NNw
__text_purify_math_cmd:N
__text_purify_math_cmd:NN
__text_purify_math_cmd:Nn
__text_purify_replace:N
__text_purify_replace_auxi:n
__text_purify_replace_auxii:n
__text_purify_expand:N
__text_purify_protect:N
__text_purify_encoding:N
__text_purify_encoding_escape:NN

As in the other parts of the module, we start off with a standard "action" loop, with expansion applied up-front.

```
\cs_new:Npn \text_purify:n #1
  {
    \__kernel_exp_not:w \exp_after:wN
      {
        \exp:w
        \exp_args:Ne \__text_purify:n
          { \text_expand:n {#1} }
      }
  }
\cs_new:Npn \__text_purify:n #1
  {
    \group_align_safe_begin:
    \__text_purify_loop:w #1
      \q__text_recursion_tail \q__text_recursion_stop
    \__text_purify_result:n { }
  }
```

As for expansion, collect up the tokens for future use.

```
\cs_new:Npn \__text_purify_store:n #1
  { \__text_purify_store:nw {#1} }
\cs_new:Npn \__text_purify_store:nw #1#2 \__text_purify_result:n #3
  { #2 \__text_purify_result:n { #3 #1 } }
\cs_new:Npn \__text_purify_end:w #1 \__text_purify_result:n #2
  {
    \group_align_safe_end:
    \exp_end:
```

```
    #2
  }
```

The main loop is a standard "tl action". Unlike the expansion or case changing, here any groups have to be run inline. Most of the business end is as before in the N-type token processing.

```
\cs_new:Npn \__text_purify_loop:w #1 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#1}
      { \__text_purify_N_type:N }
      {
        \tl_if_head_is_group:nTF {#1}
          { \__text_purify_group:n }
          { \__text_purify_space:w }
      }
    #1 \q__text_recursion_stop
  }
\cs_new:Npn \__text_purify_group:n #1 { \__text_purify_loop:w #1 }
\exp_last_unbraced:NNo \cs_new:Npn \__text_purify_space:w \c_space_tl
  {
    \__text_purify_store:n { ~ }
    \__text_purify_loop:w
  }
```

The first part of handling math mode is exactly the same as in the other functions: look for a start-of-math mode token and if found start a new loop tracking the closing token.

```
\cs_new:Npn \__text_purify_N_type:N #1
  {
    \__text_if_q_recursion_tail_stop_do:Nn #1 { \__text_purify_end:w }
    \__text_purify_N_type_aux:N #1
  }
\cs_new:Npn \__text_purify_N_type_aux:N #1
  {
    \exp_after:wN \__text_purify_math_search:NNN
      \exp_after:wN #1 \l_text_math_delims_tl
      \q__text_recursion_tail ?
      \q__text_recursion_stop
  }
\cs_new:Npn \__text_purify_math_search:NNN #1#2#3
  {
    \__text_if_q_recursion_tail_stop_do:Nn #2
      { \__text_purify_math_cmd:N #1 }
    \token_if_eq_meaning:NNTF #1 #2
      {
        \__text_use_i_delimit_by_q_recursion_stop:nw
          { \__text_purify_math_start:NNw #2 #3 }
      }
      { \__text_purify_math_search:NNN #1 }
  }
\cs_new:Npn \__text_purify_math_start:NNw #1#2#3 \q__text_recursion_stop
  {
    \__text_purify_math_loop:NNw #1#2#3 \q__text_recursion_stop
      \__text_purify_math_result:n { }
  }
\cs_new:Npn \__text_purify_math_store:n #1
```

```
  { \__text_purify_math_store:nw {#1} }
\cs_new:Npn \__text_purify_math_store:nw #1#2 \__text_purify_math_result:n #3
  { #2 \__text_purify_math_result:n { #3 #1 } }
\cs_new:Npn \__text_purify_math_end:w #1 \__text_purify_math_result:n #2
  {
    \__text_purify_store:n { $ #2 $ }
    \__text_purify_loop:w #1
  }
\cs_new:Npn \__text_purify_math_stop:Nw #1 \__text_purify_math_result:n #2
  {
    \__text_purify_store:n {#1#2}
    \__text_purify_end:w
  }
\cs_new:Npn \__text_purify_math_loop:NNw #1#2#3 \q__text_recursion_stop
  {
    \tl_if_head_is_N_type:nTF {#3}
      { \__text_purify_math_N_type:NNN }
      {
        \tl_if_head_is_group:nTF {#3}
          { \__text_purify_math_group:NNn }
          { \__text_purify_math_space:NNw }
      }
        #1#2#3 \q__text_recursion_stop
  }
\cs_new:Npn \__text_purify_math_N_type:NNN #1#2#3
  {
    \__text_if_q_recursion_tail_stop_do:Nn #3
      { \__text_purify_math_stop:Nw #1 }
    \token_if_eq_meaning:NNTF #3 #2
      { \__text_purify_math_end:w }
      {
        \__text_purify_math_store:n {#3}
        \__text_purify_math_loop:NNw #1#2
      }
  }
\cs_new:Npn \__text_purify_math_group:NNn #1#2#3
  {
    \__text_purify_math_store:n { {#3} }
    \__text_purify_math_loop:NNw #1#2
  }
\exp_after:wN \cs_new:Npn \exp_after:wN \__text_purify_math_space:NNw
  \exp_after:wN # \exp_after:wN 1
  \exp_after:wN # \exp_after:wN 2 \c_space_tl
  {
    \__text_purify_math_store:n { ~ }
    \__text_purify_math_loop:NNw #1#2
  }
```

Then handle math mode as an argument: same outcomes, different input syntax.

```
\cs_new:Npn \__text_purify_math_cmd:N #1
  {
    \exp_after:wN \__text_purify_math_cmd:NN \exp_after:wN #1
      \l_text_math_arg_tl \q__text_recursion_tail \q__text_recursion_stop
  }
\cs_new:Npn \__text_purify_math_cmd:NN #1#2
```

```
  {
    \__text_if_q_recursion_tail_stop_do:Nn #2
      { \__text_purify_replace:N #1 }
    \cs_if_eq:NNTF #2 #1
      {
        \__text_use_i_delimit_by_q_recursion_stop:nw
          { \__text_purify_math_cmd:n }
      }
      { \__text_purify_math_cmd:NN #1 }
  }
\cs_new:Npn \__text_purify_math_cmd:n #1
  { \__text_purify_math_end:w \__text_purify_math_result:n {#1} }
```

For N-type tokens, we first look for a string-context replacement before anything else: this can therefore cover anything. Assuming we don't find one, check to see if we can expand control sequences: if not, they have to be dropped. We also allow for LaTeX 2_ε `\protect`: there's an assumption that we don't have `\protect { \oops }` or similar, but that's also in the expansion code and seems like a reasonable balance.

```
\cs_new:Npn \__text_purify_replace:N #1
  {
    \bool_lazy_and:nnTF
      { \cs_if_exist_p:c { l__text_purify_ \token_to_str:N #1 _tl } }
      {
        \bool_lazy_or_p:nn
          { \token_if_cs_p:N #1 }
          { \token_if_active_p:N #1 }
      }
      {
        \exp_args:Nv \__text_purify_replace_auxi:n
          { l__text_purify_ \token_to_str:N #1 _tl }
      }
      {
        \exp_args:Ne \__text_purify_replace_auxii:n
          { \__text_token_to_explicit:N #1 }
      }
  }
\cs_new:Npn \__text_purify_replace_auxi:n #1 { \__text_purify_loop:w #1 }
\cs_new:Npn \__text_purify_replace_auxii:n #1
  {
    \token_if_cs:NTF #1
      { \__text_purify_expand:N #1 }
      {
        \__text_purify_store:n {#1}
        \__text_purify_loop:w
      }
  }
\cs_new:Npn \__text_purify_expand:N #1
  {
    \str_if_eq:nnTF {#1} { \protect }
      { \__text_purify_protect:N }
      { \__text_purify_encoding:N #1 }
  }
\cs_new:Npn \__text_purify_protect:N #1
  {
```

```
    \__text_if_q_recursion_tail_stop_do:Nn #1 { \__text_purify_end:w }
    \__text_purify_loop:w
  }
```

Handle encoding commands, as detailed for expansion.

```
\cs_new:Npn \__text_purify_encoding:N #1
  {
    \bool_lazy_or:nnTF
      { \cs_if_eq_p:NN #1 \@current@cmd }
      { \cs_if_eq_p:NN #1 \@changed@cmd }
      { \__text_purify_encoding_escape:NN }
      {
        \__text_if_expandable:NTF #1
          { \exp_after:wN \__text_purify_loop:w #1 }
          { \__text_purify_loop:w }
      }
  }
\cs_new:Npn \__text_purify_encoding_escape:NN #1#2
  {
    \__text_purify_store:n {#1}
    \__text_purify_loop:w
  }
```

(*End of definition for* `\text_purify:n` *and others. This function is documented on page 298.*)

\text_declare_purify_equivalent:Nn
\text_declare_purify_equivalent:Ne

```
\cs_new_protected:Npn \text_declare_purify_equivalent:Nn #1#2
  {
    \tl_clear_new:c { l__text_purify_ \token_to_str:N #1 _tl }
    \tl_set:cn { l__text_purify_ \token_to_str:N #1 _tl } {#2}
  }
\cs_generate_variant:Nn \text_declare_purify_equivalent:Nn { Ne }
```

(*End of definition for* `\text_declare_purify_equivalent:Nn`. *This function is documented on page 298.*)

Now pre-define a range of standard commands that need dedicated definitions in purified text. First handle font-related stuff: all of this needs to be disabled.

```
\tl_map_inline:nn
  {
    \fontencoding
    \fontfamily
    \fontseries
    \fontshape
  }
  { \text_declare_purify_equivalent:Nn #1 { \use_none:n } }
\text_declare_purify_equivalent:Nn \fontsize { \use_none:nn }
\text_declare_purify_equivalent:Nn \selectfont { }
\text_declare_purify_equivalent:Nn \usefont { \use_none:nnnn }
\tl_map_inline:nn
  {
    \emph
    \text
    \textnormal
    \textrm
    \textsf
    \texttt
```

```
      \textbf
      \textmd
      \textit
      \textsl
      \textup
      \textsc
      \textulc
    }
    { \text_declare_purify_equivalent:Nn #1 { \use:n } }
\tl_map_inline:nn
    {
      \normalfont
      \rmfamily
      \sffamily
      \ttfamily
      \bfseries
      \mdseries
      \itshape
      \scshape
      \slshape
      \upshape
      \em
      \Huge
      \LARGE
      \Large
      \footnotesize
      \huge
      \large
      \normalsize
      \scriptsize
      \small
      \tiny
    }
    { \text_declare_purify_equivalent:Nn #1 { } }
\exp_args:Nc \text_declare_purify_equivalent:Nn
    { @protected@testopt } { \use_none:nnn }
```

Environments have to be handled by pure expansion.

__text_end_env:n

```
\text_declare_purify_equivalent:Nn \begin { \use:c }
\text_declare_purify_equivalent:Nn \end { \__text_end_env:n }
\cs_new:Npn \__text_end_env:n #1 { \cs:w end #1 \cs_end: }
```

(*End of definition for* `\__text_end_env:n`.)

Some common symbols and similar ideas.

```
\text_declare_purify_equivalent:Nn \\ { }
\tl_map_inline:nn
    { \{ \} \# \$ \% \_ }
    { \text_declare_purify_equivalent:Ne #1 { \cs_to_str:N #1 } }
```

Cross-referencing.

```
\text_declare_purify_equivalent:Nn \label { \use_none:n }
```

Spaces.

```
\group_begin:
\char_set_catcode_active:N \~
\use:n
  {
    \group_end:
    \text_declare_purify_equivalent:Ne ~ { \c_space_tl }
  }
\text_declare_purify_equivalent:Nn \nobreakspace { ~ }
\text_declare_purify_equivalent:Nn \  { ~ }
\text_declare_purify_equivalent:Nn \, { ~ }
```

90.2 Accent and letter-like data for purifying text

In contrast to case changing, both 8-bit and Unicode engines need information for text purification to handle accents and letter-like functions: these all need to be removed. However, the results are of course engine-dependent.

For the letter-like commands, life is relatively easy: they are all simply added as standard exceptions. The only oddity is \SS, which gets converted to two letters. (At some stage an alternative version can presumably be added to babel or similar.)

```
\cs_set_protected:Npn \__text_loop:Nn #1#2
  {
    \quark_if_recursion_tail_stop:N #1
    \text_declare_purify_equivalent:Ne #1
      {
        \codepoint_generate:nn {"#2}
          { \char_value_catcode:n {"#2} }
      }
    \__text_loop:Nn
  }
\__text_loop:Nn
  \AA { 00C5 }
  \AE { 00C6 }
  \DH { 00D0 }
  \DJ { 0110 }
  \IJ { 0132 }
  \L  { 0141 }
  \NG { 014A }
  \O  { 00D8 }
  \OE { 0152 }
  \TH { 00DE }
  \aa { 00E5 }
  \ae { 00E6 }
  \dh { 00F0 }
  \dj { 0111 }
  \i  { 0131 }
  \j  { 0237 }
  \ij { 0132 }
  \l  { 0142 }
  \ng { 014B }
  \o  { 00F8 }
  \oe { 0153 }
```

```
  \ss { 00DF }
  \th { 00FE }
  \q_recursion_tail ?
  \q_recursion_stop
\text_declare_purify_equivalent:Nn \SS { SS }
```

__text_purify_accent:NN

Accent LICR handling is a little more complex. Accents may exist as pre-composed codepoints or as independent glyphs. The former are all saved as single token lists, whilst for the latter the combining accent needs to be re-ordered compared to the character it applies to.

```
\cs_new:Npn \__text_purify_accent:NN #1#2
  {
    \cs_if_exist:cTF
      { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
      {
        \exp_not:v
          { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
      }
      {
        \exp_not:n {#2}
        \exp_not:v { c__text_purify_ \token_to_str:N #1 _tl }
      }
  }
\tl_map_inline:nn { \` \' \^ \~ \= \u \. \" \r \H \v \d \c \k \b \t }
  { \text_declare_purify_equivalent:Nn #1 { \__text_purify_accent:NN #1 } }
```

First set up the combining accents.

```
\group_begin:
  \cs_set_protected:Npn \__text_loop:Nn #1#2
    {
      \quark_if_recursion_tail_stop:N #1
      \tl_const:ce { c__text_purify_ \token_to_str:N #1 _tl }
        { \codepoint_generate:nn {"#2} { \char_value_catcode:n { "#2 } } }
      \__text_loop:Nn
    }
  \__text_loop:Nn
    \` { 0300 }
    \' { 0301 }
    \^ { 0302 }
    \~ { 0303 }
    \= { 0304 }
    \u { 0306 }
    \. { 0307 }
    \" { 0308 }
    \r { 030A }
    \H { 030B }
    \v { 030C }
    \d { 0323 }
    \c { 0327 }
    \k { 0328 }
    \b { 0331 }
    \t { 0361 }
    \q_recursion_tail { }
    \q_recursion_stop
```

Now we handle the pre-composed accents: the list here is taken from `puenc.def`. All of the precomposed cases take a single letter as their second argument. We do not try to cover the case where an accent is added to a "real" dotless-i or -j, or a æ/Æ. Rather, we assume that if the UTF-8 character is used, it will have the real accent character too.

```
\cs_set_protected:Npn \__text_loop:NNn #1#2#3
  {
    \quark_if_recursion_tail_stop:N #1
    \tl_const:ce
      { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
      { \codepoint_generate:nn {"#3} { \char_value_catcode:n { "#3 } } }
    \__text_loop:NNn
  }
\__text_loop:NNn
  \` A   { 00C0 }
  \' A   { 00C1 }
  \^ A   { 00C2 }
  \~ A   { 00C3 }
  \" A   { 00C4 }
  \r A   { 00C5 }
  \c C   { 00C7 }
  \` E   { 00C8 }
  \' E   { 00C9 }
  \^ E   { 00CA }
  \" E   { 00CB }
  \` I   { 00CC }
  \' I   { 00CD }
  \^ I   { 00CE }
  \" I   { 00CF }
  \~ N   { 00D1 }
  \` O   { 00D2 }
  \' O   { 00D3 }
  \^ O   { 00D4 }
  \~ O   { 00D5 }
  \" O   { 00D6 }
  \` U   { 00D9 }
  \' U   { 00DA }
  \^ U   { 00DB }
  \" U   { 00DC }
  \' Y   { 00DD }
  \` a   { 00E0 }
  \' a   { 00E1 }
  \^ a   { 00E2 }
  \~ a   { 00E3 }
  \" a   { 00E4 }
  \r a   { 00E5 }
  \c c   { 00E7 }
  \` e   { 00E8 }
  \' e   { 00E9 }
  \^ e   { 00EA }
  \" e   { 00EB }
  \` i   { 00EC }
  \` \i  { 00EC }
  \' i   { 00ED }
  \' \i  { 00ED }
```

```
\^ i   { 00EE }
\^ \i  { 00EE }
\" i   { 00EF }
\" \i  { 00EF }
\~ n   { 00F1 }
\` o   { 00F2 }
\' o   { 00F3 }
\^ o   { 00F4 }
\~ o   { 00F5 }
\" o   { 00F6 }
\` u   { 00F9 }
\' u   { 00FA }
\^ u   { 00FB }
\" u   { 00FC }
\' y   { 00FD }
\" y   { 00FF }
\= A   { 0100 }
\= a   { 0101 }
\u A   { 0102 }
\u a   { 0103 }
\k A   { 0104 }
\k a   { 0105 }
\' C   { 0106 }
\' c   { 0107 }
\^ C   { 0108 }
\^ c   { 0109 }
\. C   { 010A }
\. c   { 010B }
\v C   { 010C }
\v c   { 010D }
\v D   { 010E }
\v d   { 010F }
\= E   { 0112 }
\= e   { 0113 }
\u E   { 0114 }
\u e   { 0115 }
\. E   { 0116 }
\. e   { 0117 }
\k E   { 0118 }
\k e   { 0119 }
\v E   { 011A }
\v e   { 011B }
\^ G   { 011C }
\^ g   { 011D }
\u G   { 011E }
\u g   { 011F }
\. G   { 0120 }
\. g   { 0121 }
\c G   { 0122 }
\c g   { 0123 }
\^ H   { 0124 }
\^ h   { 0125 }
\~ I   { 0128 }
\~ i   { 0129 }
```

```
\~ \i  { 0129 }
\= I   { 012A }
\= i   { 012B }
\= \i  { 012B }
\u I   { 012C }
\u i   { 012D }
\u \i  { 012D }
\k I   { 012E }
\k i   { 012F }
\k \i  { 012F }
\. I   { 0130 }
\^ J   { 0134 }
\^ j   { 0135 }
\^ \j  { 0135 }
\c K   { 0136 }
\c k   { 0137 }
\' L   { 0139 }
\' l   { 013A }
\c L   { 013B }
\c l   { 013C }
\v L   { 013D }
\v l   { 013E }
\. L   { 013F }
\. l   { 0140 }
\' N   { 0143 }
\' n   { 0144 }
\c N   { 0145 }
\c n   { 0146 }
\v N   { 0147 }
\v n   { 0148 }
\= O   { 014C }
\= o   { 014D }
\u O   { 014E }
\u o   { 014F }
\H O   { 0150 }
\H o   { 0151 }
\' R   { 0154 }
\' r   { 0155 }
\c R   { 0156 }
\c r   { 0157 }
\v R   { 0158 }
\v r   { 0159 }
\' S   { 015A }
\' s   { 015B }
\^ S   { 015C }
\^ s   { 015D }
\c S   { 015E }
\c s   { 015F }
\v S   { 0160 }
\v s   { 0161 }
\c T   { 0162 }
\c t   { 0163 }
\v T   { 0164 }
\v t   { 0165 }
```

```
\~ U  { 0168 }
\~ u  { 0169 }
\= U  { 016A }
\= u  { 016B }
\u U  { 016C }
\u u  { 016D }
\r U  { 016E }
\r u  { 016F }
\H U  { 0170 }
\H u  { 0171 }
\k U  { 0172 }
\k u  { 0173 }
\^ W  { 0174 }
\^ w  { 0175 }
\^ Y  { 0176 }
\^ y  { 0177 }
\" Y  { 0178 }
\' Z  { 0179 }
\' z  { 017A }
\. Z  { 017B }
\. z  { 017C }
\v Z  { 017D }
\v z  { 017E }
\v A  { 01CD }
\v a  { 01CE }
\v I  { 01CF }
\v \i { 01D0 }
\v i  { 01D0 }
\v O  { 01D1 }
\v o  { 01D2 }
\v U  { 01D3 }
\v u  { 01D4 }
\v G  { 01E6 }
\v g  { 01E7 }
\v K  { 01E8 }
\v k  { 01E9 }
\k O  { 01EA }
\k o  { 01EB }
\v \j { 01F0 }
\v j  { 01F0 }
\' G  { 01F4 }
\' g  { 01F5 }
\` N  { 01F8 }
\` n  { 01F9 }
\' \AE { 01FC }
\' \ae { 01FD }
\' \O { 01FE }
\' \o { 01FF }
\v H  { 021E }
\v h  { 021F }
\. A  { 0226 }
\. a  { 0227 }
\c E  { 0228 }
\c e  { 0229 }
```

```
    \. O   { 022E }
    \. o   { 022F }
    \= Y   { 0232 }
    \= y   { 0233 }
    \q_recursion_tail ? { }
    \q_recursion_stop
\group_end:
```

(*End of definition for* `\__text_purify_accent:NN`.)

```
⟨/package⟩
```

Chapter 91

l3box implementation

```
⟨*package⟩
⟨@@=box⟩
```

91.1 Support code

__box_dim_eval:w
__box_dim_eval:n

Evaluating a dimension expression expandably. The only difference with `\dim_eval:n` is the lack of `\dim_use:N`, to produce an internal dimension rather than expand it into characters.

```
\cs_new_eq:NN \__box_dim_eval:w \tex_dimexpr:D
\cs_new:Npn \__box_dim_eval:n #1
  { \__box_dim_eval:w #1 \scan_stop: }
```

(*End of definition for* `\__box_dim_eval:w` *and* `\__box_dim_eval:n`*.*)

__kernel_kern:n

We need kerns in a few places. At present, we don't have a module for this concept, so it goes in at first use: here. The idea is to avoid repeated use of the bare primitive.

```
\cs_new_protected:Npn \__kernel_kern:n #1
  { \tex_kern:D \__box_dim_eval:n {#1} }
```

(*End of definition for* `\__kernel_kern:n`*.*)

91.2 Creating and initialising boxes

The following test files are used for this code: m3box001.lvt*.*

\box_new:N
\box_new:c

Defining a new ⟨*box*⟩ register: remember that box 255 is not generally available.

```
\cs_new_protected:Npn \box_new:N #1
  {
    \__kernel_chk_if_free_cs:N #1
    \cs:w newbox \cs_end: #1
  }
\cs_generate_variant:Nn \box_new:N { c }
```

Clear a ⟨*box*⟩ register.

\box_clear:N
\box_clear:c
\box_gclear:N
\box_gclear:c

```
\cs_new_protected:Npn \box_clear:N #1
  { \box_set_eq:NN  #1 \c_empty_box }
\cs_new_protected:Npn \box_gclear:N #1
  { \box_gset_eq:NN #1 \c_empty_box }
\cs_generate_variant:Nn \box_clear:N  { c }
\cs_generate_variant:Nn \box_gclear:N { c }
```

Clear or new.

\box_clear_new:N
\box_clear_new:c
\box_gclear_new:N
\box_gclear_new:c

```
\cs_new_protected:Npn \box_clear_new:N #1
  { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
\cs_new_protected:Npn \box_gclear_new:N #1
  { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
\cs_generate_variant:Nn \box_clear_new:N  { c }
\cs_generate_variant:Nn \box_gclear_new:N { c }
```

Assigning the contents of a box to be another box.

\box_set_eq:NN
\box_set_eq:cN
\box_set_eq:Nc
\box_set_eq:cc
\box_gset_eq:NN
\box_gset_eq:cN
\box_gset_eq:Nc
\box_gset_eq:cc

```
\cs_new_protected:Npn \box_set_eq:NN #1#2
  { \tex_setbox:D #1 \tex_copy:D #2 }
\cs_new_protected:Npn \box_gset_eq:NN #1#2
  { \tex_global:D \tex_setbox:D #1 \tex_copy:D #2 }
\cs_generate_variant:Nn \box_set_eq:NN  { c , Nc , cc }
\cs_generate_variant:Nn \box_gset_eq:NN { c , Nc , cc }
```

Assigning the contents of a box to be another box, then drops the original box.

\box_set_eq_drop:NN
\box_set_eq_drop:cN
\box_set_eq_drop:Nc
\box_set_eq_drop:cc
\box_gset_eq_drop:NN
\box_gset_eq_drop:cN
\box_gset_eq_drop:Nc
\box_gset_eq_drop:cc

```
\cs_new_protected:Npn \box_set_eq_drop:NN #1#2
  { \tex_setbox:D #1 \tex_box:D #2 }
\cs_new_protected:Npn \box_gset_eq_drop:NN #1#2
  { \tex_global:D \tex_setbox:D #1 \tex_box:D #2 }
\cs_generate_variant:Nn \box_set_eq_drop:NN  { c , Nc , cc }
\cs_generate_variant:Nn \box_gset_eq_drop:NN { c , Nc , cc }
```

Copies of the cs functions defined in l3basics.

\box_if_exist_p:N
\box_if_exist_p:c
\box_if_exist:N*TF*
\box_if_exist:c*TF*

```
\prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N
  { TF , T , F , p }
\prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c
  { TF , T , F , p }
```

91.3 Measuring and setting box dimensions

Accessing the height, depth, and width of a ⟨*box*⟩ register.

\box_ht:N
\box_ht:c
\box_dp:N
\box_dp:c
\box_wd:N
\box_wd:c

```
\cs_new_eq:NN \box_ht:N \tex_ht:D
\cs_new_eq:NN \box_dp:N \tex_dp:D
\cs_new_eq:NN \box_wd:N \tex_wd:D
\cs_generate_variant:Nn \box_ht:N { c }
\cs_generate_variant:Nn \box_dp:N { c }
\cs_generate_variant:Nn \box_wd:N { c }
```

\box_ht_plus_dp:N

The `\box_ht:N` and `\box_dp:N` primitives do not expand but rather are suitable for use after `\the` or inside dimension expressions. Here we obtain the same behaviour by using `\__box_dim_eval:n` (basically `\dimexpr`) rather than `\dim_eval:n` (basically `\the \dimexpr`).

```
\cs_new_protected:Npn \box_ht_plus_dp:N #1
  { \__box_dim_eval:n { \box_ht:N #1 + \box_dp:N #1 } }
\cs_generate_variant:Nn \box_ht_plus_dp:N { c }
```

Setting the size whilst respecting local scope requires copying; the same issue does not come up when working globally. When debugging, the dimension expression #2 is surrounded by parentheses to catch early termination.

\box_set_ht:Nn
\box_set_ht:cn
\box_gset_ht:Nn
\box_gset_ht:cn
\box_set_dp:Nn
\box_set_dp:cn
\box_gset_dp:Nn
\box_gset_dp:cn
\box_set_wd:Nn
\box_set_wd:cn
\box_gset_wd:Nn
\box_gset_wd:cn

```
\cs_new_protected:Npn \box_set_dp:Nn #1#2
  {
    \tex_setbox:D #1 = \tex_copy:D #1
    \box_dp:N #1 \__box_dim_eval:n {#2}
  }
\cs_generate_variant:Nn \box_set_dp:Nn { c }
\cs_new_protected:Npn \box_gset_dp:Nn #1#2
  { \box_dp:N #1 \__box_dim_eval:n {#2} }
\cs_generate_variant:Nn \box_gset_dp:Nn { c }
\cs_new_protected:Npn \box_set_ht:Nn #1#2
  {
    \tex_setbox:D #1 = \tex_copy:D #1
    \box_ht:N #1 \__box_dim_eval:n {#2}
  }
\cs_generate_variant:Nn \box_set_ht:Nn { c }
\cs_new_protected:Npn \box_gset_ht:Nn #1#2
  { \box_ht:N #1 \__box_dim_eval:n {#2} }
\cs_generate_variant:Nn \box_gset_ht:Nn { c }
\cs_new_protected:Npn \box_set_wd:Nn #1#2
  {
    \tex_setbox:D #1 = \tex_copy:D #1
    \box_wd:N #1 \__box_dim_eval:n {#2}
  }
\cs_generate_variant:Nn \box_set_wd:Nn { c }
\cs_new_protected:Npn \box_gset_wd:Nn #1#2
  { \box_wd:N #1 \__box_dim_eval:n {#2} }
\cs_generate_variant:Nn \box_gset_wd:Nn { c }
```

91.4 Using boxes

Using a ⟨*box*⟩. These are just TeX primitives with meaningful names.

\box_use_drop:N
\box_use_drop:c
\box_use:N
\box_use:c

```
\cs_new_eq:NN \box_use_drop:N \tex_box:D
\cs_new_eq:NN \box_use:N \tex_copy:D
\cs_generate_variant:Nn \box_use_drop:N { c }
\cs_generate_variant:Nn \box_use:N { c }
```

Move box material in different directions. When debugging, the dimension expression #1 is surrounded by parentheses to catch early termination.

\box_move_left:nn
\box_move_right:nn
\box_move_up:nn
\box_move_down:nn

```
\cs_new_protected:Npn \box_move_left:nn #1#2
  { \tex_moveleft:D \__box_dim_eval:n {#1} #2 }
\cs_new_protected:Npn \box_move_right:nn #1#2
  { \tex_moveright:D \__box_dim_eval:n {#1} #2 }
\cs_new_protected:Npn \box_move_up:nn #1#2
  { \tex_raise:D \__box_dim_eval:n {#1} #2 }
\cs_new_protected:Npn \box_move_down:nn #1#2
  { \tex_lower:D \__box_dim_eval:n {#1} #2 }
```

91.5 Box conditionals

\if_hbox:N
\if_vbox:N
\if_box_empty:N

The primitives for testing if a ⟨*box*⟩ is empty/void or which type of box it is.

```
\cs_new_eq:NN \if_hbox:N      \tex_ifhbox:D
\cs_new_eq:NN \if_vbox:N      \tex_ifvbox:D
\cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D
```

\box_if_horizontal_p:N
\box_if_horizontal_p:c
\box_if_horizontal:N*TF*
\box_if_horizontal:c*TF*
\box_if_vertical_p:N
\box_if_vertical_p:c
\box_if_vertical:N*TF*
\box_if_vertical:c*TF*

```
\prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
  { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
  { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\prg_generate_conditional_variant:Nnn \box_if_horizontal:N
  { c } { p , T , F , TF }
\prg_generate_conditional_variant:Nnn \box_if_vertical:N
  { c } { p , T , F , TF }
```

Testing if a ⟨*box*⟩ is empty/void.

\box_if_empty_p:N
\box_if_empty_p:c
\box_if_empty:N*TF*
\box_if_empty:c*TF*

```
\prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
  { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\prg_generate_conditional_variant:Nnn \box_if_empty:N
  { c } { p , T , F , TF }
```

(*End of definition for* `\box_new:N` *and others. These functions are documented on page 301.*)

91.6 The last box inserted

\box_set_to_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c

Set a box to the previous box.

```
\cs_new_protected:Npn \box_set_to_last:N #1
  { \tex_setbox:D #1 \tex_lastbox:D }
\cs_new_protected:Npn \box_gset_to_last:N #1
  { \tex_global:D \tex_setbox:D #1 \tex_lastbox:D }
\cs_generate_variant:Nn \box_set_to_last:N  { c }
\cs_generate_variant:Nn \box_gset_to_last:N { c }
```

(*End of definition for* `\box_set_to_last:N` *and* `\box_gset_to_last:N`*. These functions are documented on page 304.*)

91.7 Constant boxes

\c_empty_box

A box we never use.

```
\box_new:N \c_empty_box
```

(*End of definition for* `\c_empty_box`*. This variable is documented on page 304.*)

91.8 Scratch boxes

\l_tmpa_box
\l_tmpb_box
\g_tmpa_box
\g_tmpb_box

Scratch boxes.

```
\box_new:N \l_tmpa_box
\box_new:N \l_tmpb_box
\box_new:N \g_tmpa_box
\box_new:N \g_tmpb_box
```

(*End of definition for* `\l_tmpa_box` *and others. These variables are documented on page 304.*)

91.9 Viewing box contents

TEX's `\showbox` is not really that helpful in many cases, and it is also inconsistent with other LATEX3 `show` functions as it does not actually shows material in the terminal. So we provide a richer set of functionality.

\box_show:N
\box_show:c
\box_show:Nnn
\box_show:cnn

Essentially a wrapper around the internal function, but evaluating the breadth and depth arguments now outside the group.

```
\cs_new_protected:Npn \box_show:N #1
  { \box_show:Nnn #1 \c_max_int \c_max_int }
\cs_generate_variant:Nn \box_show:N { c }
\cs_new_protected:Npn \box_show:Nnn #1#2#3
  { \__box_show:NNff 1 #1 { \int_eval:n {#2} } { \int_eval:n {#3} } }
\cs_generate_variant:Nn \box_show:Nnn { c }
```

(*End of definition for* `\box_show:N` *and* `\box_show:Nnn`*. These functions are documented on page 305.*)

\box_log:N
\box_log:c
\box_log:Nnn
\box_log:cnn
__box_log:nNnn

Getting TEX to write to the log without interruption the run is done by altering the interaction mode. For that, the ε-TEX extensions are needed.

```
\cs_new_protected:Npn \box_log:N #1
  { \box_log:Nnn #1 \c_max_int \c_max_int }
\cs_generate_variant:Nn \box_log:N { c }
\cs_new_protected:Npn \box_log:Nnn
  { \exp_args:No \__box_log:nNnn { \tex_the:D \tex_interactionmode:D } }
\cs_new_protected:Npn \__box_log:nNnn #1#2#3#4
  {
    \int_gset:Nn \tex_interactionmode:D { 0 }
    \__box_show:NNff 0 #2 { \int_eval:n {#3} } { \int_eval:n {#4} }
    \int_gset:Nn \tex_interactionmode:D {#1}
  }
\cs_generate_variant:Nn \box_log:Nnn { c }
```

(*End of definition for* `\box_log:N`*,* `\box_log:Nnn`*, and* `\__box_log:nNnn`*. These functions are documented on page 305.*)

__box_show:NNnn
__box_show:NNff

The internal auxiliary to actually do the output uses a group to deal with breadth and depth values. The `\use:n` here gives better output appearance. Setting `\tracingonline` and `\errorcontextlines` is used to control what appears in the terminal.

```
\cs_new_protected:Npn \__box_show:NNnn #1#2#3#4
  {
    \box_if_exist:NTF #2
      {
        \group_begin:
```

```
          \int_set:Nn \tex_showboxbreadth:D {#3}
          \int_set:Nn \tex_showboxdepth:D   {#4}
          \int_set:Nn \tex_tracingonline:D  {#1}
          \int_set:Nn \tex_errorcontextlines:D { -1 }
          \tex_showbox:D \use:n {#2}
        \group_end:
      }
      {
        \msg_error:nne { kernel } { variable-not-defined }
          { \token_to_str:N #2 }
      }
  }
\cs_generate_variant:Nn \__box_show:NNnn { NNff }
```

(*End of definition for* `\__box_show:NNnn`.)

91.10 Horizontal mode boxes

`\hbox:n`

(*The test suite for this command, and others in this file, is* m3box002.lvt.)

Put a horizontal box directly into the input stream.

```
\cs_new_protected:Npn \hbox:n #1
  { \tex_hbox:D \scan_stop: { \color_group_begin: #1 \color_group_end: } }
```

(*End of definition for* `\hbox:n`. *This function is documented on page 305.*)

`\hbox_set:Nn`
`\hbox_set:cn`
`\hbox_gset:Nn`
`\hbox_gset:cn`

```
\cs_new_protected:Npn \hbox_set:Nn #1#2
  {
    \tex_setbox:D #1 \tex_hbox:D
      { \color_group_begin: #2 \color_group_end: }
  }
\cs_new_protected:Npn \hbox_gset:Nn #1#2
  {
    \tex_global:D \tex_setbox:D #1 \tex_hbox:D
      { \color_group_begin: #2 \color_group_end: }
  }
\cs_generate_variant:Nn \hbox_set:Nn { c }
\cs_generate_variant:Nn \hbox_gset:Nn { c }
```

(*End of definition for* `\hbox_set:Nn` *and* `\hbox_gset:Nn`. *These functions are documented on page 305.*)

`\hbox_set_to_wd:Nnn`
`\hbox_set_to_wd:cnn`
`\hbox_gset_to_wd:Nnn`
`\hbox_gset_to_wd:cnn`

Storing material in a horizontal box with a specified width. Again, put the dimension expression in parentheses when debugging.

```
\cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
  {
    \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
      { \color_group_begin: #3 \color_group_end: }
  }
\cs_new_protected:Npn \hbox_gset_to_wd:Nnn #1#2#3
  {
    \tex_global:D \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
      { \color_group_begin: #3 \color_group_end: }
  }
\cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
\cs_generate_variant:Nn \hbox_gset_to_wd:Nnn { c }
```

(*End of definition for* `\hbox_set_to_wd:Nnn` *and* `\hbox_gset_to_wd:Nnn`*. These functions are documented on page* 306*.*)

`\hbox_set:Nw`
`\hbox_set:cw`
`\hbox_gset:Nw`
`\hbox_gset:cw`
`\hbox_set_end:`
`\hbox_gset_end:`

Storing material in a horizontal box. This type is useful in environment definitions.

```
\cs_new_protected:Npn \hbox_set:Nw  #1
  {
    \tex_setbox:D #1 \tex_hbox:D
      \c_group_begin_token
        \color_group_begin:
  }
\cs_new_protected:Npn \hbox_gset:Nw  #1
  {
    \tex_global:D \tex_setbox:D #1 \tex_hbox:D
      \c_group_begin_token
        \color_group_begin:
  }
\cs_generate_variant:Nn \hbox_set:Nw  { c }
\cs_generate_variant:Nn \hbox_gset:Nw { c }
\cs_new_protected:Npn \hbox_set_end:
  {
      \color_group_end:
    \c_group_end_token
  }
\cs_new_eq:NN \hbox_gset_end: \hbox_set_end:
```

(*End of definition for* `\hbox_set:Nw` *and others. These functions are documented on page* 306*.*)

`\hbox_set_to_wd:Nnw`
`\hbox_set_to_wd:cnw`
`\hbox_gset_to_wd:Nnw`
`\hbox_gset_to_wd:cnw`

Combining the above ideas.

```
\cs_new_protected:Npn \hbox_set_to_wd:Nnw #1#2
  {
    \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
      \c_group_begin_token
        \color_group_begin:
  }
\cs_new_protected:Npn \hbox_gset_to_wd:Nnw #1#2
  {
    \tex_global:D \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
      \c_group_begin_token
        \color_group_begin:
  }
\cs_generate_variant:Nn \hbox_set_to_wd:Nnw  { c }
\cs_generate_variant:Nn \hbox_gset_to_wd:Nnw { c }
```

(*End of definition for* `\hbox_set_to_wd:Nnw` *and* `\hbox_gset_to_wd:Nnw`*. These functions are documented on page* 306*.*)

`\hbox_to_wd:nn`
`\hbox_to_zero:n`

Put a horizontal box directly into the input stream.

```
\cs_new_protected:Npn \hbox_to_wd:nn #1#2
  {
    \tex_hbox:D to \__box_dim_eval:n {#1}
      { \color_group_begin: #2 \color_group_end: }
  }
\cs_new_protected:Npn \hbox_to_zero:n #1
  {
    \tex_hbox:D to \c_zero_dim
```

```
      { \color_group_begin: #1 \color_group_end: }
  }
```

(*End of definition for* `\hbox_to_wd:nn` *and* `\hbox_to_zero:n`. *These functions are documented on page* 305.)

`\hbox_overlap_center:n`
`\hbox_overlap_left:n`
`\hbox_overlap_right:n`

Put a zero-sized box with the contents pushed against one side (which makes it stick out on the other) directly into the input stream.

```
\cs_new_protected:Npn \hbox_overlap_center:n  #1
  { \hbox_to_zero:n { \tex_hss:D #1 \tex_hss:D } }
\cs_new_protected:Npn \hbox_overlap_left:n  #1
  { \hbox_to_zero:n { \tex_hss:D #1 } }
\cs_new_protected:Npn \hbox_overlap_right:n #1
  { \hbox_to_zero:n { #1 \tex_hss:D } }
```

(*End of definition for* `\hbox_overlap_center:n`, `\hbox_overlap_left:n`, *and* `\hbox_overlap_right:n`. *These functions are documented on page* 306.)

`\hbox_unpack:N`
`\hbox_unpack:c`
`\hbox_unpack_drop:N`
`\hbox_unpack_drop:c`

Unpacking a box and if requested also clear it.

```
\cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
\cs_new_eq:NN \hbox_unpack_drop:N \tex_unhbox:D
\cs_generate_variant:Nn \hbox_unpack:N { c }
\cs_generate_variant:Nn \hbox_unpack_drop:N { c }
```

(*End of definition for* `\hbox_unpack:N` *and* `\hbox_unpack_drop:N`. *These functions are documented on page* 306.)

91.11 Vertical mode boxes

TeX ends these boxes directly with the internal *end_graf* routine. This means that there is no `\par` at the end of vertical boxes unless we insert one. Thus all vertical boxes include a `\par` just before closing the color group.

`\vbox:n`

The following test files are used for this code: m3box003.lvt.

`\vbox_top:n`

The following test files are used for this code: m3box003.lvt.

Put a vertical box directly into the input stream.

```
\cs_new_protected:Npn \vbox:n #1
  { \tex_vbox:D { \color_group_begin: #1 \par \color_group_end: } }
\cs_new_protected:Npn \vbox_top:n #1
  { \tex_vtop:D { \color_group_begin: #1 \par \color_group_end: } }
```

(*End of definition for* `\vbox:n` *and* `\vbox_top:n`. *These functions are documented on page* 307.)

`\vbox_to_ht:nn`
`\vbox_to_zero:n`
`\vbox_to_ht:nn`
`\vbox_to_zero:n`

Put a vertical box directly into the input stream.

```
\cs_new_protected:Npn \vbox_to_ht:nn #1#2
  {
    \tex_vbox:D to \__box_dim_eval:n {#1}
      { \color_group_begin: #2 \par \color_group_end: }
  }
\cs_new_protected:Npn \vbox_to_zero:n #1
  {
    \tex_vbox:D to \c_zero_dim
      { \color_group_begin: #1 \par \color_group_end: }
  }
```

(*End of definition for* `\vbox_to_ht:nn` *and others. These functions are documented on page 307.*)

`\vbox_set:Nn`
`\vbox_set:cn`
`\vbox_gset:Nn`
`\vbox_gset:cn`

Storing material in a vertical box with a natural height.

```
\cs_new_protected:Npn \vbox_set:Nn #1#2
  {
    \tex_setbox:D #1 \tex_vbox:D
      { \color_group_begin: #2 \par \color_group_end: }
  }
\cs_new_protected:Npn \vbox_gset:Nn #1#2
  {
    \tex_global:D \tex_setbox:D #1 \tex_vbox:D
      { \color_group_begin: #2 \par \color_group_end: }
  }
\cs_generate_variant:Nn \vbox_set:Nn  { c }
\cs_generate_variant:Nn \vbox_gset:Nn { c }
```

(*End of definition for* `\vbox_set:Nn` *and* `\vbox_gset:Nn`*. These functions are documented on page 307.*)

`\vbox_set_top:Nn`
`\vbox_set_top:cn`
`\vbox_gset_top:Nn`
`\vbox_gset_top:cn`

Storing material in a vertical box with a natural height and reference point at the baseline of the first object in the box.

```
\cs_new_protected:Npn \vbox_set_top:Nn #1#2
  {
    \tex_setbox:D #1 \tex_vtop:D
      { \color_group_begin: #2 \par \color_group_end: }
  }
\cs_new_protected:Npn \vbox_gset_top:Nn #1#2
  {
    \tex_global:D \tex_setbox:D #1 \tex_vtop:D
      { \color_group_begin: #2 \par \color_group_end: }
  }
\cs_generate_variant:Nn \vbox_set_top:Nn { c }
\cs_generate_variant:Nn \vbox_gset_top:Nn { c }
```

(*End of definition for* `\vbox_set_top:Nn` *and* `\vbox_gset_top:Nn`*. These functions are documented on page 307.*)

`\vbox_set_to_ht:Nnn`
`\vbox_set_to_ht:cnn`
`\vbox_gset_to_ht:Nnn`
`\vbox_gset_to_ht:cnn`

Storing material in a vertical box with a specified height.

```
\cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3
  {
    \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}
      { \color_group_begin: #3 \par \color_group_end: }
  }
\cs_new_protected:Npn \vbox_gset_to_ht:Nnn #1#2#3
  {
    \tex_global:D \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}
      { \color_group_begin: #3 \par \color_group_end: }
  }
\cs_generate_variant:Nn \vbox_set_to_ht:Nnn  { c }
\cs_generate_variant:Nn \vbox_gset_to_ht:Nnn { c }
```

(*End of definition for* `\vbox_set_to_ht:Nnn` *and* `\vbox_gset_to_ht:Nnn`*. These functions are documented on page 307.*)

\vbox_set:Nw
\vbox_set:cw
\vbox_gset:Nw
\vbox_gset:cw
\vbox_set_end:
\vbox_gset_end:

Storing material in a vertical box. This type is useful in environment definitions.

```
\cs_new_protected:Npn \vbox_set:Nw #1
  {
    \tex_setbox:D #1 \tex_vbox:D
      \c_group_begin_token
        \color_group_begin:
  }
\cs_new_protected:Npn \vbox_gset:Nw #1
  {
    \tex_global:D \tex_setbox:D #1 \tex_vbox:D
      \c_group_begin_token
        \color_group_begin:
  }
\cs_generate_variant:Nn \vbox_set:Nw  { c }
\cs_generate_variant:Nn \vbox_gset:Nw { c }
\cs_new_protected:Npn \vbox_set_end:
  {
        \par
      \color_group_end:
    \c_group_end_token
  }
\cs_new_eq:NN \vbox_gset_end: \vbox_set_end:
```

(*End of definition for* `\vbox_set:Nw` *and others. These functions are documented on page 307.*)

\vbox_set_to_ht:Nnw
\vbox_set_to_ht:cnw
\vbox_gset_to_ht:Nnw
\vbox_gset_to_ht:cnw

A combination of the above ideas.

```
\cs_new_protected:Npn \vbox_set_to_ht:Nnw #1#2
  {
    \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}
      \c_group_begin_token
        \color_group_begin:
  }
\cs_new_protected:Npn \vbox_gset_to_ht:Nnw #1#2
  {
    \tex_global:D \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}
      \c_group_begin_token
        \color_group_begin:
  }
\cs_generate_variant:Nn \vbox_set_to_ht:Nnw  { c }
\cs_generate_variant:Nn \vbox_gset_to_ht:Nnw { c }
```

(*End of definition for* `\vbox_set_to_ht:Nnw` *and* `\vbox_gset_to_ht:Nnw`*. These functions are documented on page 308.*)

\vbox_unpack:N
\vbox_unpack:c
\vbox_unpack_drop:N
\vbox_unpack_drop:c

Unpacking a box and if requested also clear it.

```
\cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
\cs_new_eq:NN \vbox_unpack_drop:N \tex_unvbox:D
\cs_generate_variant:Nn \vbox_unpack:N { c }
\cs_generate_variant:Nn \vbox_unpack_drop:N { c }
```

(*End of definition for* `\vbox_unpack:N` *and* `\vbox_unpack_drop:N`*. These functions are documented on page 308.*)

\vbox_set_split_to_ht:NNn
\vbox_set_split_to_ht:cNn
\vbox_set_split_to_ht:Ncn
\vbox_set_split_to_ht:ccn
\vbox_gset_split_to_ht:NNn
\vbox_gset_split_to_ht:cNn
\vbox_gset_split_to_ht:Ncn
\vbox_gset_split_to_ht:ccn

Splitting a vertical box in two.

```
\cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
  { \tex_setbox:D #1 \tex_vsplit:D #2 to \__box_dim_eval:n {#3} }
\cs_generate_variant:Nn \vbox_set_split_to_ht:NNn { c , Nc , cc }
\cs_new_protected:Npn \vbox_gset_split_to_ht:NNn #1#2#3
  {
    \tex_global:D \tex_setbox:D #1
      \tex_vsplit:D #2 to \__box_dim_eval:n {#3}
  }
\cs_generate_variant:Nn \vbox_gset_split_to_ht:NNn { c , Nc , cc }
```

(*End of definition for* \vbox_set_split_to_ht:NNn *and* \vbox_gset_split_to_ht:NNn*. These functions are documented on page 308.*)

91.12 Affine transformations

\l__box_angle_fp

When rotating boxes, the angle itself may be needed by the engine-dependent code. This is done using the fp module so that the value is tidied up properly.

```
\fp_new:N \l__box_angle_fp
```

(*End of definition for* \l__box_angle_fp*.*)

\l__box_cos_fp
\l__box_sin_fp

These are used to hold the calculated sine and cosine values while carrying out a rotation.

```
\fp_new:N \l__box_cos_fp
\fp_new:N \l__box_sin_fp
```

(*End of definition for* \l__box_cos_fp *and* \l__box_sin_fp*.*)

\l__box_top_dim
\l__box_bottom_dim
\l__box_left_dim
\l__box_right_dim

These are the positions of the four edges of a box before manipulation.

```
\dim_new:N \l__box_top_dim
\dim_new:N \l__box_bottom_dim
\dim_new:N \l__box_left_dim
\dim_new:N \l__box_right_dim
```

(*End of definition for* \l__box_top_dim *and others.*)

\l__box_top_new_dim
\l__box_bottom_new_dim
\l__box_left_new_dim
\l__box_right_new_dim

These are the positions of the four edges of a box after manipulation.

```
\dim_new:N \l__box_top_new_dim
\dim_new:N \l__box_bottom_new_dim
\dim_new:N \l__box_left_new_dim
\dim_new:N \l__box_right_new_dim
```

(*End of definition for* \l__box_top_new_dim *and others.*)

\l__box_internal_box

Scratch space, but also needed by some parts of the driver.

```
\box_new:N \l__box_internal_box
```

(*End of definition for* \l__box_internal_box*.*)

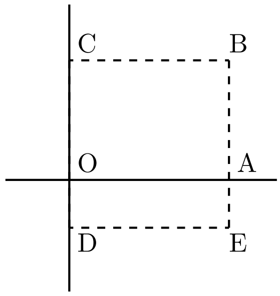

Figure 1: Coordinates of a box prior to rotation.

\box_rotate:Nn
\box_rotate:cn
\box_grotate:Nn
\box_grotate:cn
__box_rotate:NnN
__box_rotate:N
__box_rotate_xdir:nnN
__box_rotate_ydir:nnN
__box_rotate_quadrant_one:
__box_rotate_quadrant_two:
__box_rotate_quadrant_three:
__box_rotate_quadrant_four:

Rotation of a box starts with working out the relevant sine and cosine. The actual rotation is in an auxiliary to keep the flow slightly clearer

```
\cs_new_protected:Npn \box_rotate:Nn #1#2
  { \__box_rotate:NnN #1 {#2} \hbox_set:Nn }
\cs_generate_variant:Nn \box_rotate:Nn { c }
\cs_new_protected:Npn \box_grotate:Nn #1#2
  { \__box_rotate:NnN #1 {#2} \hbox_gset:Nn }
\cs_generate_variant:Nn \box_grotate:Nn { c }
\cs_new_protected:Npn \__box_rotate:NnN #1#2#3
  {
    #3 #1
      {
        \fp_set:Nn \l__box_angle_fp {#2}
        \fp_set:Nn \l__box_sin_fp { sind ( \l__box_angle_fp ) }
        \fp_set:Nn \l__box_cos_fp { cosd ( \l__box_angle_fp ) }
        \__box_rotate:N #1
      }
  }
```

The edges of the box are then recorded: the left edge is always at zero. Rotation of the four edges then takes place: this is most efficiently done on a quadrant by quadrant basis.

```
\cs_new_protected:Npn \__box_rotate:N #1
  {
    \dim_set:Nn \l__box_top_dim    {  \box_ht:N #1 }
    \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
    \dim_set:Nn \l__box_right_dim  {  \box_wd:N #1 }
    \dim_zero:N \l__box_left_dim
```

The next step is to work out the x and y coordinates of vertices of the rotated box in relation to its original coordinates. The box can be visualized with vertices B, C, D and E is illustrated (Figure 1). The vertex O is the reference point on the baseline, and in this implementation is also the centre of rotation. The formulae are, for a point P and angle α:

$$
\begin{aligned}
P'_x &= P_x - O_x \\
P'_y &= P_y - O_y \\
P''_x &= (P'_x \cos(\alpha)) - (P'_y \sin(\alpha)) \\
P''_y &= (P'_x \sin(\alpha)) + (P'_y \cos(\alpha)) \\
P'''_x &= P''_x + O_x + L_x \\
P'''_y &= P''_y + O_y
\end{aligned}
$$

The “extra” horizontal translation L_x at the end is calculated so that the leftmost point of the resulting box has x-coordinate 0. This is desirable as TEX boxes must have the reference point at the left edge of the box. (As O is always $(0, 0)$, this part of the calculation is omitted here.)

```
    \fp_compare:nNnTF \l__box_sin_fp > \c_zero_fp
      {
        \fp_compare:nNnTF \l__box_cos_fp > \c_zero_fp
          { \__box_rotate_quadrant_one: }
          { \__box_rotate_quadrant_two: }
      }
      {
        \fp_compare:nNnTF \l__box_cos_fp < \c_zero_fp
          { \__box_rotate_quadrant_three: }
          { \__box_rotate_quadrant_four: }
      }
```

The position of the box edges are now known, but the box at this stage be misplaced relative to the current TEX reference point. So the content of the box is moved such that the reference point of the rotated box is in the same place as the original.

```
    \hbox_set:Nn \l__box_internal_box { \box_use:N #1 }
    \hbox_set:Nn \l__box_internal_box
      {
        \__kernel_kern:n { -\l__box_left_new_dim }
        \hbox:n
          {
            \__box_backend_rotate:Nn
              \l__box_internal_box
              \l__box_angle_fp
          }
      }
```

Tidy up the size of the box so that the material is actually inside the bounding box. The result can then be used to reset the original box.

```
    \box_set_ht:Nn \l__box_internal_box {  \l__box_top_new_dim }
    \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
    \box_set_wd:Nn \l__box_internal_box
      { \l__box_right_new_dim - \l__box_left_new_dim }
    \box_use_drop:N \l__box_internal_box
  }
```

These functions take a general point (#1, #2) and rotate its location about the origin, using the previously-set sine and cosine values. Each function gives only one component of the location of the updated point. This is because for rotation of a box each step needs only one value, and so performance is gained by avoiding working out both x' and y' at the same time. Contrast this with the equivalent function in the l3coffins module, where both parts are needed.

```
\cs_new_protected:Npn \__box_rotate_xdir:nnN #1#2#3
  {
    \dim_set:Nn #3
      {
        \fp_to_dim:n
          {
              \l__box_cos_fp * \dim_to_fp:n {#1}
            - \l__box_sin_fp * \dim_to_fp:n {#2}
```

```
        }
    }
  }
\cs_new_protected:Npn \__box_rotate_ydir:nnN #1#2#3
  {
    \dim_set:Nn #3
      {
        \fp_to_dim:n
          {
              \l__box_sin_fp * \dim_to_fp:n {#1}
            + \l__box_cos_fp * \dim_to_fp:n {#2}
          }
      }
  }
```

Rotation of the edges is done using a different formula for each quadrant. In every case, the top and bottom edges only need the resulting y-values, whereas the left and right edges need the x-values. Each case is a question of picking out which corner ends up at with the maximum top, bottom, left and right value. Doing this by hand means a lot less calculating and avoids lots of comparisons.

```
\cs_new_protected:Npn \__box_rotate_quadrant_one:
  {
    \__box_rotate_ydir:nnN \l__box_right_dim \l__box_top_dim
      \l__box_top_new_dim
    \__box_rotate_ydir:nnN \l__box_left_dim  \l__box_bottom_dim
      \l__box_bottom_new_dim
    \__box_rotate_xdir:nnN \l__box_left_dim  \l__box_top_dim
      \l__box_left_new_dim
    \__box_rotate_xdir:nnN \l__box_right_dim \l__box_bottom_dim
      \l__box_right_new_dim
  }
\cs_new_protected:Npn \__box_rotate_quadrant_two:
  {
    \__box_rotate_ydir:nnN \l__box_right_dim \l__box_bottom_dim
      \l__box_top_new_dim
    \__box_rotate_ydir:nnN \l__box_left_dim  \l__box_top_dim
      \l__box_bottom_new_dim
    \__box_rotate_xdir:nnN \l__box_right_dim  \l__box_top_dim
      \l__box_left_new_dim
    \__box_rotate_xdir:nnN \l__box_left_dim   \l__box_bottom_dim
      \l__box_right_new_dim
  }
\cs_new_protected:Npn \__box_rotate_quadrant_three:
  {
    \__box_rotate_ydir:nnN \l__box_left_dim  \l__box_bottom_dim
      \l__box_top_new_dim
    \__box_rotate_ydir:nnN \l__box_right_dim \l__box_top_dim
      \l__box_bottom_new_dim
    \__box_rotate_xdir:nnN \l__box_right_dim \l__box_bottom_dim
      \l__box_left_new_dim
    \__box_rotate_xdir:nnN \l__box_left_dim   \l__box_top_dim
      \l__box_right_new_dim
  }
\cs_new_protected:Npn \__box_rotate_quadrant_four:
```

```
  {
    \__box_rotate_ydir:nnN \l__box_left_dim  \l__box_top_dim
      \l__box_top_new_dim
    \__box_rotate_ydir:nnN \l__box_right_dim \l__box_bottom_dim
      \l__box_bottom_new_dim
    \__box_rotate_xdir:nnN \l__box_left_dim  \l__box_bottom_dim
      \l__box_left_new_dim
    \__box_rotate_xdir:nnN \l__box_right_dim \l__box_top_dim
      \l__box_right_new_dim
  }
```

(*End of definition for* `\box_rotate:Nn` *and others. These functions are documented on page* 312.)

`\l__box_scale_x_fp`
`\l__box_scale_y_fp`

Scaling is potentially-different in the two axes.

```
\fp_new:N \l__box_scale_x_fp
\fp_new:N \l__box_scale_y_fp
```

(*End of definition for* `\l__box_scale_x_fp` *and* `\l__box_scale_y_fp`.)

`\box_resize_to_wd_and_ht_plus_dp:Nnn`
`\box_resize_to_wd_and_ht_plus_dp:cnn`
`\box_gresize_to_wd_and_ht_plus_dp:Nnn`
`\box_gresize_to_wd_and_ht_plus_dp:cnn`
`\__box_resize_to_wd_and_ht_plus_dp:NnnN`
`\__box_resize_set_corners:N`
`\__box_resize:N`
`\__box_resize:NNN`

Resizing a box starts by working out the various dimensions of the existing box.

```
\cs_new_protected:Npn \box_resize_to_wd_and_ht_plus_dp:Nnn #1#2#3
  {
    \__box_resize_to_wd_and_ht_plus_dp:NnnN #1 {#2} {#3}
      \hbox_set:Nn
  }
\cs_generate_variant:Nn \box_resize_to_wd_and_ht_plus_dp:Nnn { c }
\cs_new_protected:Npn \box_gresize_to_wd_and_ht_plus_dp:Nnn #1#2#3
  {
    \__box_resize_to_wd_and_ht_plus_dp:NnnN #1 {#2} {#3}
      \hbox_gset:Nn
  }
\cs_generate_variant:Nn \box_gresize_to_wd_and_ht_plus_dp:Nnn { c }
\cs_new_protected:Npn \__box_resize_to_wd_and_ht_plus_dp:NnnN #1#2#3#4
  {
    #4 #1
      {
        \__box_resize_set_corners:N #1
```

The x-scaling and resulting box size is easy enough to work out: the dimension is that given as `#2`, and the scale is simply the new width divided by the old one.

```
        \fp_set:Nn \l__box_scale_x_fp
          { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
```

The y-scaling needs both the height and the depth of the current box.

```
        \fp_set:Nn \l__box_scale_y_fp
          {
              \dim_to_fp:n {#3}
            / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
          }
```

Hand off to the auxiliary which does the rest of the work.

```
        \__box_resize:N #1
      }
  }
\cs_new_protected:Npn \__box_resize_set_corners:N #1
  {
```

```
    \dim_set:Nn \l__box_top_dim    {  \box_ht:N #1 }
    \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
    \dim_set:Nn \l__box_right_dim  {  \box_wd:N #1 }
    \dim_zero:N \l__box_left_dim
  }
```

With at least one real scaling to do, the next phase is to find the new edge coordinates. In the x direction this is relatively easy: just scale the right edge. In the y direction, both dimensions have to be scaled, and this again needs the absolute scale value. Once that is all done, the common resize/rescale code can be employed.

```
\cs_new_protected:Npn \__box_resize:N #1
  {
    \__box_resize:NNN \l__box_right_new_dim
      \l__box_scale_x_fp \l__box_right_dim
    \__box_resize:NNN \l__box_bottom_new_dim
      \l__box_scale_y_fp \l__box_bottom_dim
    \__box_resize:NNN \l__box_top_new_dim
      \l__box_scale_y_fp \l__box_top_dim
    \__box_resize_common:N #1
  }
\cs_new_protected:Npn \__box_resize:NNN #1#2#3
  {
    \dim_set:Nn #1
      { \fp_to_dim:n { \fp_abs:n { #2 } * \dim_to_fp:n { #3 } } }
  }
```

(*End of definition for* `\box_resize_to_wd_and_ht_plus_dp:Nnn` *and others. These functions are documented on page 311.*)

\box_resize_to_ht:Nn
\box_resize_to_ht:cn
\box_gresize_to_ht:Nn
\box_gresize_to_ht:cn
__box_resize_to_ht:NnN
\box_resize_to_ht_plus_dp:Nn
\box_resize_to_ht_plus_dp:cn
\box_gresize_to_ht_plus_dp:Nn
\box_gresize_to_ht_plus_dp:cn
__box_resize_to_ht_plus_dp:NnN
\box_resize_to_wd:Nn
\box_resize_to_wd:cn
\box_gresize_to_wd:Nn
\box_gresize_to_wd:cn
__box_resize_to_wd:NnN
\box_resize_to_wd_and_ht:Nnn
\box_resize_to_wd_and_ht:cnn
\box_gresize_to_wd_and_ht:Nnn
\box_gresize_to_wd_and_ht:cnn
__box_resize_to_wd_ht:NnnN

Scaling to a (total) height or to a width is a simplified version of the main resizing operation, with the scale simply copied between the two parts. The internal auxiliary is called using the scaling value twice, as the sign for both parts is needed (as this allows the same internal code to be used as for the general case).

```
\cs_new_protected:Npn \box_resize_to_ht:Nn #1#2
  { \__box_resize_to_ht:NnN #1 {#2} \hbox_set:Nn }
\cs_generate_variant:Nn \box_resize_to_ht:Nn { c }
\cs_new_protected:Npn \box_gresize_to_ht:Nn #1#2
  { \__box_resize_to_ht:NnN #1 {#2} \hbox_gset:Nn }
\cs_generate_variant:Nn \box_gresize_to_ht:Nn { c }
\cs_new_protected:Npn \__box_resize_to_ht:NnN #1#2#3
  {
    #3 #1
      {
        \__box_resize_set_corners:N #1
        \fp_set:Nn \l__box_scale_y_fp
          {
              \dim_to_fp:n {#2}
            / \dim_to_fp:n { \l__box_top_dim }
          }
        \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
        \__box_resize:N #1
      }
  }
\cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2
```

```
  { \__box_resize_to_ht_plus_dp:NnN #1 {#2} \hbox_set:Nn }
\cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
\cs_new_protected:Npn \box_gresize_to_ht_plus_dp:Nn #1#2
  { \__box_resize_to_ht_plus_dp:NnN #1 {#2} \hbox_gset:Nn }
\cs_generate_variant:Nn \box_gresize_to_ht_plus_dp:Nn { c }
\cs_new_protected:Npn \__box_resize_to_ht_plus_dp:NnN #1#2#3
  {
    #3 #1
      {
        \__box_resize_set_corners:N #1
        \fp_set:Nn \l__box_scale_y_fp
          {
              \dim_to_fp:n {#2}
            / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
          }
        \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
        \__box_resize:N #1
      }
  }
\cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
  { \__box_resize_to_wd:NnN #1 {#2} \hbox_set:Nn }
\cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
\cs_new_protected:Npn \box_gresize_to_wd:Nn #1#2
  { \__box_resize_to_wd:NnN #1 {#2} \hbox_gset:Nn }
\cs_generate_variant:Nn \box_gresize_to_wd:Nn { c }
\cs_new_protected:Npn \__box_resize_to_wd:NnN #1#2#3
  {
    #3 #1
      {
        \__box_resize_set_corners:N #1
        \fp_set:Nn \l__box_scale_x_fp
          { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
        \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
        \__box_resize:N #1
      }
  }
\cs_new_protected:Npn \box_resize_to_wd_and_ht:Nnn #1#2#3
  { \__box_resize_to_wd_and_ht:NnnN #1 {#2} {#3} \hbox_set:Nn }
\cs_generate_variant:Nn \box_resize_to_wd_and_ht:Nnn { c }
\cs_new_protected:Npn \box_gresize_to_wd_and_ht:Nnn #1#2#3
  { \__box_resize_to_wd_and_ht:NnnN #1 {#2} {#3} \hbox_gset:Nn }
\cs_generate_variant:Nn \box_gresize_to_wd_and_ht:Nnn { c }
\cs_new_protected:Npn \__box_resize_to_wd_and_ht:NnnN #1#2#3#4
  {
    #4 #1
      {
        \__box_resize_set_corners:N #1
        \fp_set:Nn \l__box_scale_x_fp
          { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
        \fp_set:Nn \l__box_scale_y_fp
          {
              \dim_to_fp:n {#3}
            / \dim_to_fp:n { \l__box_top_dim }
          }
```

```
        \__box_resize:N #1
      }
  }
```

(*End of definition for* `\box_resize_to_ht:Nn` *and others. These functions are documented on page 310.*)

`\box_scale:Nnn`
`\box_scale:cnn`
`\box_gscale:Nnn`
`\box_gscale:cnn`
`\__box_scale:NnnN`
`\__box_scale:N`

When scaling a box, setting the scaling itself is easy enough. The new dimensions are also relatively easy to find, allowing only for the need to keep them positive in all cases. Once that is done then after a check for the trivial scaling a hand-off can be made to the common code. The code here is split into two as this allows sharing with the auto-resizing functions.

```
\cs_new_protected:Npn \box_scale:Nnn #1#2#3
  { \__box_scale:NnnN #1 {#2} {#3} \hbox_set:Nn }
\cs_generate_variant:Nn \box_scale:Nnn { c }
\cs_new_protected:Npn \box_gscale:Nnn #1#2#3
  { \__box_scale:NnnN #1 {#2} {#3} \hbox_gset:Nn }
\cs_generate_variant:Nn \box_gscale:Nnn { c }
\cs_new_protected:Npn \__box_scale:NnnN #1#2#3#4
  {
    #4 #1
      {
        \fp_set:Nn \l__box_scale_x_fp {#2}
        \fp_set:Nn \l__box_scale_y_fp {#3}
        \__box_scale:N #1
      }
  }
\cs_new_protected:Npn \__box_scale:N #1
  {
    \dim_set:Nn \l__box_top_dim    {  \box_ht:N #1 }
    \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
    \dim_set:Nn \l__box_right_dim  {  \box_wd:N #1 }
    \dim_zero:N \l__box_left_dim
    \dim_set:Nn \l__box_top_new_dim
      { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
    \dim_set:Nn \l__box_bottom_new_dim
      { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
    \dim_set:Nn \l__box_right_new_dim
      { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
    \__box_resize_common:N #1
  }
```

(*End of definition for* `\box_scale:Nnn` *and others. These functions are documented on page 312.*)

`\box_autosize_to_wd_and_ht:Nnn`
`\box_autosize_to_wd_and_ht:cnn`
`\box_gautosize_to_wd_and_ht:Nnn`
`\box_gautosize_to_wd_and_ht:cnn`
`\box_autosize_to_wd_and_ht_plus_dp:Nnn`
`\box_autosize_to_wd_and_ht_plus_dp:cnn`
`\box_gautosize_to_wd_and_ht_plus_dp:Nnn`
`\box_gautosize_to_wd_and_ht_plus_dp:cnn`
`\__box_autosize:NnnnN`

Although autosizing a box uses dimensions, it has more in common in implementation with scaling. As such, most of the real work here is done elsewhere.

```
\cs_new_protected:Npn \box_autosize_to_wd_and_ht:Nnn #1#2#3
  { \__box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 } \hbox_set:Nn }
\cs_generate_variant:Nn \box_autosize_to_wd_and_ht:Nnn { c }
\cs_new_protected:Npn \box_gautosize_to_wd_and_ht:Nnn #1#2#3
  { \__box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 } \hbox_gset:Nn }
\cs_generate_variant:Nn \box_gautosize_to_wd_and_ht:Nnn { c }
\cs_new_protected:Npn \box_autosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
  {
    \__box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 }
```

```
      \hbox_set:Nn
  }
\cs_generate_variant:Nn \box_autosize_to_wd_and_ht_plus_dp:Nnn { c }
\cs_new_protected:Npn \box_gautosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
  {
    \__box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 }
      \hbox_gset:Nn
  }
\cs_generate_variant:Nn \box_gautosize_to_wd_and_ht_plus_dp:Nnn { c }
\cs_new_protected:Npn \__box_autosize:NnnnN #1#2#3#4#5
  {
    #5 #1
      {
        \fp_set:Nn \l__box_scale_x_fp { ( \dim_to_fp:n {#2} ) / \box_wd:N #1 }
        \fp_set:Nn \l__box_scale_y_fp
          { (  \dim_to_fp:n {#3} ) / ( \dim_to_fp:n {#4} ) }
        \fp_compare:nNnTF \l__box_scale_x_fp > \l__box_scale_y_fp
          { \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp }
          { \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp }
        \__box_scale:N #1
      }
  }
```

(*End of definition for* `\box_autosize_to_wd_and_ht:Nnn` *and others. These functions are documented on page 310.*)

`\__box_resize_common:N` The main resize function places its input into a box which start off with zero width, and includes the handles for engine rescaling.

```
\cs_new_protected:Npn \__box_resize_common:N #1
  {
    \hbox_set:Nn \l__box_internal_box
      {
        \__box_backend_scale:Nnn
          #1
          \l__box_scale_x_fp
          \l__box_scale_y_fp
      }
```

The new height and depth can be applied directly.

```
    \fp_compare:nNnTF \l__box_scale_y_fp > \c_zero_fp
      {
        \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
        \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
      }
      {
        \box_set_dp:Nn \l__box_internal_box { \l__box_top_new_dim }
        \box_set_ht:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
      }
```

Things are not quite as obvious for the width, as the reference point needs to remain unchanged. For positive scaling factors resizing the box is all that is needed. However, for case of a negative scaling the material must be shifted such that the reference point ends up in the right place.

```
    \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
      {
```

```
        \hbox_to_wd:nn { \l__box_right_new_dim }
          {
            \__kernel_kern:n { \l__box_right_new_dim }
            \box_use_drop:N \l__box_internal_box
            \tex_hss:D
          }
      }
      {
        \box_set_wd:Nn \l__box_internal_box { \l__box_right_new_dim }
        \hbox:n
          {
            \__kernel_kern:n { 0pt }
            \box_use_drop:N \l__box_internal_box
            \tex_hss:D
          }
      }
  }
```

(*End of definition for* `\__box_resize_common:N`.)

91.13 Viewing part of a box

`\box_set_clipped:N`
`\box_set_clipped:c`
`\box_gset_clipped:N`
`\box_gset_clipped:c`

A wrapper around the driver-dependent code.

```
\cs_new_protected:Npn \box_set_clipped:N #1
  { \hbox_set:Nn #1 { \__box_backend_clip:N #1 } }
\cs_generate_variant:Nn \box_set_clipped:N { c }
\cs_new_protected:Npn \box_gset_clipped:N #1
  { \hbox_gset:Nn #1 { \__box_backend_clip:N #1 } }
\cs_generate_variant:Nn \box_gset_clipped:N { c }
```

(*End of definition for* `\box_set_clipped:N` *and* `\box_gset_clipped:N`. *These functions are documented on page 312.*)

`\box_set_trim:Nnnnn`
`\box_set_trim:cnnnn`
`\box_gset_trim:Nnnnn`
`\box_gset_trim:cnnnn`
`\__box_set_trim:NnnnnN`

Trimming from the left- and right-hand edges of the box is easy: kern the appropriate parts off each side.

```
\cs_new_protected:Npn \box_set_trim:Nnnnn #1#2#3#4#5
  { \__box_set_trim:NnnnnN #1 {#2} {#3} {#4} {#5} \box_set_eq:NN }
\cs_generate_variant:Nn \box_set_trim:Nnnnn { c }
\cs_new_protected:Npn \box_gset_trim:Nnnnn #1#2#3#4#5
  { \__box_set_trim:NnnnnN #1 {#2} {#3} {#4} {#5} \box_gset_eq:NN }
\cs_generate_variant:Nn \box_gset_trim:Nnnnn { c }
\cs_new_protected:Npn \__box_set_trim:NnnnnN #1#2#3#4#5#6
  {
    \hbox_set:Nn \l__box_internal_box
      {
        \__kernel_kern:n { -#2 }
        \box_use:N #1
        \__kernel_kern:n { -#4 }
      }
```

For the height and depth, there is a need to watch the baseline is respected. Material always has to stay on the correct side, so trimming has to check that there is enough material to trim. First, the bottom edge. If there is enough depth, simply set the depth, or if not move down so the result is zero depth. `\box_move_down:nn` is used in both cases so the resulting box always contains a \lower primitive. The internal box is used here as it allows safe use of \box_set_dp:Nn.

```
    \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_down:nn \c_zero_dim
              { \box_use_drop:N \l__box_internal_box }
          }
        \box_set_dp:Nn \l__box_internal_box { \box_dp:N #1 - (#3) }
      }
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_down:nn { (#3) - \box_dp:N #1 }
              { \box_use_drop:N \l__box_internal_box }
          }
        \box_set_dp:Nn \l__box_internal_box \c_zero_dim
      }
```

Same thing, this time from the top of the box.

```
    \dim_compare:nNnTF { \box_ht:N \l__box_internal_box } > {#5}
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_up:nn \c_zero_dim
              { \box_use_drop:N \l__box_internal_box }
          }
        \box_set_ht:Nn \l__box_internal_box
          { \box_ht:N \l__box_internal_box - (#5) }
      }
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_up:nn { (#5) - \box_ht:N \l__box_internal_box }
              { \box_use_drop:N \l__box_internal_box }
          }
        \box_set_ht:Nn \l__box_internal_box \c_zero_dim
      }
    #6 #1 \l__box_internal_box
  }
```

(*End of definition for* \box_set_trim:Nnnnn *,* \box_gset_trim:Nnnnn *, and* __box_set_trim:NnnnnN*. These functions are documented on page* *312**.*)

\box_set_viewport:Nnnnn
\box_set_viewport:cnnnn
\box_gset_viewport:Nnnnn
\box_gset_viewport:cnnnn
__box_viewport:NnnnnN

The same general logic as for the trim operation, but with absolute dimensions. As a result, there are some things to watch out for in the vertical direction.

```
\cs_new_protected:Npn \box_set_viewport:Nnnnn #1#2#3#4#5
  { \__box_set_viewport:NnnnnN #1 {#2} {#3} {#4} {#5} \box_set_eq:NN }
\cs_generate_variant:Nn \box_set_viewport:Nnnnn { c }
\cs_new_protected:Npn \box_gset_viewport:Nnnnn #1#2#3#4#5
  { \__box_set_viewport:NnnnnN #1 {#2} {#3} {#4} {#5} \box_gset_eq:NN }
\cs_generate_variant:Nn \box_gset_viewport:Nnnnn { c }
\cs_new_protected:Npn \__box_set_viewport:NnnnnN #1#2#3#4#5#6
```

```
  {
    \hbox_set:Nn \l__box_internal_box
      {
        \__kernel_kern:n { -#2 }
        \box_use:N #1
        \__kernel_kern:n { #4 - \box_wd:N #1 }
      }
    \dim_compare:nNnTF {#3} < \c_zero_dim
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_down:nn \c_zero_dim
              { \box_use_drop:N \l__box_internal_box }
          }
        \box_set_dp:Nn \l__box_internal_box { - \__box_dim_eval:n {#3} }
      }
      {
        \hbox_set:Nn \l__box_internal_box
          { \box_move_down:nn {#3} { \box_use_drop:N \l__box_internal_box } }
        \box_set_dp:Nn \l__box_internal_box \c_zero_dim
      }
    \dim_compare:nNnTF {#5} > \c_zero_dim
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_up:nn \c_zero_dim
              { \box_use_drop:N \l__box_internal_box }
          }
        \box_set_ht:Nn \l__box_internal_box
          {
            (#5)
            \dim_compare:nNnT {#3} > \c_zero_dim
              { - (#3) }
          }
      }
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_up:nn { - \__box_dim_eval:n {#5} }
              { \box_use_drop:N \l__box_internal_box }
          }
        \box_set_ht:Nn \l__box_internal_box \c_zero_dim
      }
    #6 #1 \l__box_internal_box
  }
```

(*End of definition for* `\box_set_viewport:Nnnnn`*,* `\box_gset_viewport:Nnnnn`*, and* `\__box_viewport:NnnnnN`*. These functions are documented on page 312.*)

⟨/package⟩

Chapter 92

l3coffins implementation

```
⟨*package⟩
⟨@@=coffin⟩
```

92.1 Coffins: data structures and general variables

\l__coffin_internal_box
\l__coffin_internal_dim
\l__coffin_internal_tl

Scratch variables.

```
\box_new:N \l__coffin_internal_box
\dim_new:N \l__coffin_internal_dim
\tl_new:N  \l__coffin_internal_tl
```

(*End of definition for* `\l__coffin_internal_box`*,* `\l__coffin_internal_dim`*, and* `\l__coffin_internal_tl`*.*)

\c__coffin_corners_prop

The "corners"; of a coffin define the real content, as opposed to the TeX bounding box. They all start off in the same place, of course.

```
\prop_const_from_keyval:Nn \c__coffin_corners_prop
  {
    tl = { 0pt } { 0pt } ,
    tr = { 0pt } { 0pt } ,
    bl = { 0pt } { 0pt } ,
    br = { 0pt } { 0pt } ,
  }
```

(*End of definition for* `\c__coffin_corners_prop`*.*)

\c__coffin_poles_prop

Pole positions are given for horizontal, vertical and reference-point based values.

```
\prop_const_from_keyval:Nn \c__coffin_poles_prop
  {
    l  = { 0pt } { 0pt } { 0pt } { 1000pt } ,
    hc = { 0pt } { 0pt } { 0pt } { 1000pt } ,
    r  = { 0pt } { 0pt } { 0pt } { 1000pt } ,
    b  = { 0pt } { 0pt } { 1000pt } { 0pt } ,
    vc = { 0pt } { 0pt } { 1000pt } { 0pt } ,
    t  = { 0pt } { 0pt } { 1000pt } { 0pt } ,
    B  = { 0pt } { 0pt } { 1000pt } { 0pt } ,
    H  = { 0pt } { 0pt } { 1000pt } { 0pt } ,
    T  = { 0pt } { 0pt } { 1000pt } { 0pt } ,
  }
```

(*End of definition for* `\c__coffin_poles_prop`*.*)

`\l__coffin_slope_A_fp`
`\l__coffin_slope_B_fp`

Used for calculations of intersections.

```
\fp_new:N \l__coffin_slope_A_fp
\fp_new:N \l__coffin_slope_B_fp
```

(*End of definition for* `\l__coffin_slope_A_fp` *and* `\l__coffin_slope_B_fp`*.*)

`\l__coffin_error_bool`

For propagating errors so that parts of the code can work around them.

```
\bool_new:N \l__coffin_error_bool
```

(*End of definition for* `\l__coffin_error_bool`*.*)

`\l__coffin_offset_x_dim`
`\l__coffin_offset_y_dim`

The offset between two sets of coffin handles when typesetting. These values are corrected from those requested in an alignment for the positions of the handles.

```
\dim_new:N \l__coffin_offset_x_dim
\dim_new:N \l__coffin_offset_y_dim
```

(*End of definition for* `\l__coffin_offset_x_dim` *and* `\l__coffin_offset_y_dim`*.*)

`\l__coffin_pole_a_tl`
`\l__coffin_pole_b_tl`

Needed for finding the intersection of two poles.

```
\tl_new:N \l__coffin_pole_a_tl
\tl_new:N \l__coffin_pole_b_tl
```

(*End of definition for* `\l__coffin_pole_a_tl` *and* `\l__coffin_pole_b_tl`*.*)

`\l__coffin_x_dim`
`\l__coffin_y_dim`
`\l__coffin_x_prime_dim`
`\l__coffin_y_prime_dim`

For calculating intersections and so forth.

```
\dim_new:N \l__coffin_x_dim
\dim_new:N \l__coffin_y_dim
\dim_new:N \l__coffin_x_prime_dim
\dim_new:N \l__coffin_y_prime_dim
```

(*End of definition for* `\l__coffin_x_dim` *and others.*)

92.2 Basic coffin functions

There are a number of basic functions needed for creating coffins and placing material in them. This all relies on the following data structures.

`\__coffin_to_value:N`

Coffins are a two-part structure and we rely on the internal nature of box allocation to make everything work. As such, we need an interface to turn coffin identifiers into numbers. For the purposes here, the signature allowed is `N` despite the nature of the underlying primitive.

```
\cs_new_eq:NN \__coffin_to_value:N \tex_number:D
```

(*End of definition for* `\__coffin_to_value:N`*.*)

\coffin_if_exist_p:N
\coffin_if_exist_p:c
\coffin_if_exist:NTF
\coffin_if_exist:cTF

Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. A cleaner way to handle this is provided here: both the box and the coffin structure are checked.

```
\prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }
  {
    \cs_if_exist:NTF #1
      {
        \cs_if_exist:cTF { coffin ~ \__coffin_to_value:N #1 ~ poles }
          { \prg_return_true: }
          { \prg_return_false: }
      }
      { \prg_return_false: }
  }
\prg_generate_conditional_variant:Nnn \coffin_if_exist:N
  { c } { p , T , F , TF }
```

(*End of definition for* `\coffin_if_exist:NTF`. *This function is documented on page 314.*)

__coffin_if_exist:NT

Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. So a wrapper is provided to deal with this correctly, issuing an error on erroneous use.

```
\cs_new_protected:Npn \__coffin_if_exist:NT #1#2
  {
    \coffin_if_exist:NTF #1
      { #2 }
      {
        \msg_error:nne { coffin } { unknown }
          { \token_to_str:N #1 }
      }
  }
```

(*End of definition for* `\__coffin_if_exist:NT`.)

\coffin_clear:N
\coffin_clear:c
\coffin_gclear:N
\coffin_gclear:c

Clearing coffins means emptying the box and resetting all of the structures.

```
\cs_new_protected:Npn \coffin_clear:N #1
  {
    \__coffin_if_exist:NT #1
      {
        \box_clear:N #1
        \__coffin_reset_structure:N #1
      }
  }
\cs_generate_variant:Nn \coffin_clear:N { c }
\cs_new_protected:Npn \coffin_gclear:N #1
  {
    \__coffin_if_exist:NT #1
      {
        \box_gclear:N #1
        \__coffin_greset_structure:N #1
      }
  }
\cs_generate_variant:Nn \coffin_gclear:N { c }
```

(*End of definition for* `\coffin_clear:N` *and* `\coffin_gclear:N`. *These functions are documented on page 314.*)

\coffin_new:N
\coffin_new:c

Creating a new coffin means making the underlying box and adding the data structures. The \debug_suspend: and \debug_resume: functions prevent \prop_gclear_-new:c from writing useless information to the log file.

```
\cs_new_protected:Npn \coffin_new:N #1
  {
    \box_new:N #1
    \debug_suspend:
    \prop_gclear_new:c { coffin ~ \__coffin_to_value:N #1 ~ corners }
    \prop_gclear_new:c { coffin ~ \__coffin_to_value:N #1 ~ poles }
    \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
      \c__coffin_corners_prop
    \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
      \c__coffin_poles_prop
    \debug_resume:
  }
\cs_generate_variant:Nn \coffin_new:N { c }
```

(*End of definition for* `\coffin_new:N`. *This function is documented on page 314.*)

\hcoffin_set:Nn
\hcoffin_set:cn
\hcoffin_gset:Nn
\hcoffin_gset:cn

Horizontal coffins are relatively easy: set the appropriate box, reset the structures then update the handle positions.

```
\cs_new_protected:Npn \hcoffin_set:Nn #1#2
  {
    \__coffin_if_exist:NT #1
      {
        \hbox_set:Nn #1
          {
            \color_ensure_current:
            #2
          }
        \coffin_reset_poles:N #1
      }
  }
\cs_generate_variant:Nn \hcoffin_set:Nn { c }
\cs_new_protected:Npn \hcoffin_gset:Nn #1#2
  {
    \__coffin_if_exist:NT #1
      {
        \hbox_gset:Nn #1
          {
            \color_ensure_current:
            #2
          }
        \coffin_greset_poles:N #1
      }
  }
\cs_generate_variant:Nn \hcoffin_gset:Nn { c }
```

(*End of definition for* `\hcoffin_set:Nn` *and* `\hcoffin_gset:Nn`. *These functions are documented on page 315.*)

\vcoffin_set:Nnn
\vcoffin_set:cnn
\vcoffin_gset:Nnn
\vcoffin_gset:cnn
__coffin_set_vertical:NnnNNN
__coffin_set_vertical_aux:

Setting vertical coffins is more complex. First, the material is typeset with a given width. The default handles and poles are set as for a horizontal coffin, before finding the top baseline using a temporary box. No \color_ensure_current: here as that would add a whatsit to the start of the vertical box and mess up the location of the `T` pole (see *TeX by Topic* for discussion of the `\vtop` primitive, used to do the measuring).

```
\cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
  {
    \__coffin_set_vertical:NnnNNN #1 {#2} {#3}
      \vbox_set:Nn \coffin_reset_poles:N \__coffin_set_pole:Nnn
  }
\cs_generate_variant:Nn \vcoffin_set:Nnn { c }
\cs_new_protected:Npn \vcoffin_gset:Nnn #1#2#3
  {
    \__coffin_set_vertical:NnnNNN #1 {#2} {#3}
      \vbox_gset:Nn \coffin_greset_poles:N \__coffin_gset_pole:Nnn
  }
\cs_generate_variant:Nn \vcoffin_gset:Nnn { c }
\cs_new_protected:Npn \__coffin_set_vertical:NnnNNN #1#2#3#4#5#6
  {
    \__coffin_if_exist:NT #1
      {
        #4 #1
          {
            \dim_set:Nn \tex_hsize:D {#2}
            \__coffin_set_vertical_aux:
            #3
          }
        #5 #1
        \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
        #6 #1 { T }
          {
            { 0pt }
            {
              \dim_eval:n
                { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
            }
            { 1000pt }
            { 0pt }
          }
        \box_clear:N \l__coffin_internal_box
      }
  }
\cs_new_protected:Npe \__coffin_set_vertical_aux:
  {
    \bool_lazy_and:nnT
      { \cs_if_exist_p:N \fmtname }
      { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
      {
        \dim_set_eq:NN \exp_not:N \linewidth \tex_hsize:D
        \dim_set_eq:NN \exp_not:N \columnwidth \tex_hsize:D
      }
  }
```

(*End of definition for* `\vcoffin_set:Nnn` *and others. These functions are documented on page* *315*.)

`\hcoffin_set:Nw`
`\hcoffin_set:cw`
`\hcoffin_gset:Nw`
`\hcoffin_gset:cw`
`\hcoffin_set_end:`
`\hcoffin_gset_end:`

These are the "begin"/"end" versions of the above: watch the grouping!

```
\cs_new_protected:Npn \hcoffin_set:Nw #1
```

```
  {
    \__coffin_if_exist:NT #1
      {
        \hbox_set:Nw #1 \color_ensure_current:
          \cs_set_protected:Npn \hcoffin_set_end:
            {
              \hbox_set_end:
              \coffin_reset_poles:N #1
            }
      }
  }
\cs_generate_variant:Nn \hcoffin_set:Nw { c }
\cs_new_protected:Npn \hcoffin_gset:Nw #1
  {
    \__coffin_if_exist:NT #1
      {
        \hbox_gset:Nw #1 \color_ensure_current:
          \cs_set_protected:Npn \hcoffin_gset_end:
            {
              \hbox_gset_end:
              \coffin_greset_poles:N #1
            }
      }
  }
\cs_generate_variant:Nn \hcoffin_gset:Nw { c }
\cs_new_protected:Npn \hcoffin_set_end: { }
\cs_new_protected:Npn \hcoffin_gset_end: { }
```

(*End of definition for* `\hcoffin_set:Nw` *and others. These functions are documented on page 315.*)

`\vcoffin_set:Nw`
`\vcoffin_set:cnw`
`\vcoffin_gset:Nnw`
`\vcoffin_gset:cnw`
`\__coffin_set_vertical:NnNNNNNw`
`\vcoffin_set_end:`
`\vcoffin_gset_end:`

The same for vertical coffins.

```
\cs_new_protected:Npn \vcoffin_set:Nnw #1#2
  {
    \__coffin_set_vertical:NnNNNNNw #1 {#2} \vbox_set:Nw
      \vcoffin_set_end:
      \vbox_set_end: \coffin_reset_poles:N \__coffin_set_pole:Nnn
  }
\cs_generate_variant:Nn \vcoffin_set:Nnw { c }
\cs_new_protected:Npn \vcoffin_gset:Nnw #1#2
  {
    \__coffin_set_vertical:NnNNNNNw #1 {#2} \vbox_gset:Nw
      \vcoffin_gset_end:
      \vbox_gset_end: \coffin_greset_poles:N \__coffin_gset_pole:Nnn
  }
\cs_generate_variant:Nn \vcoffin_gset:Nnw { c }
\cs_new_protected:Npn \__coffin_set_vertical:NnNNNNNw #1#2#3#4#5#6#7
  {
    \__coffin_if_exist:NT #1
      {
        #3 #1
          \dim_set:Nn \tex_hsize:D {#2}
          \__coffin_set_vertical_aux:
          \cs_set_protected:Npn #4
            {
```

```
                #5
                #6 #1
                \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
                #7 #1 { T }
                  {
                    { 0pt }
                    {
                      \dim_eval:n
                        { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
                    }
                    { 1000pt }
                    { 0pt }
                  }
                \box_clear:N \l__coffin_internal_box
              }
        }
    }
\cs_new_protected:Npn \vcoffin_set_end: { }
\cs_new_protected:Npn \vcoffin_gset_end: { }
```

(*End of definition for* `\vcoffin_set:Nnw` *and others. These functions are documented on page 315.*)

`\coffin_set_eq:NN` `\coffin_set_eq:Nc` `\coffin_set_eq:cN` `\coffin_set_eq:cc` `\coffin_gset_eq:NN` `\coffin_gset_eq:Nc` `\coffin_gset_eq:cN` `\coffin_gset_eq:cc`

Setting two coffins equal is just a wrapper around other functions.

```
\cs_new_protected:Npn \coffin_set_eq:NN #1#2
  {
    \__coffin_if_exist:NT #2
      {
        \box_set_eq:NN #1 #2
        \prop_set_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ corners }
          { coffin ~ \__coffin_to_value:N #2 ~ corners }
        \prop_set_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ poles }
          { coffin ~ \__coffin_to_value:N #2 ~ poles }
      }
  }
\cs_generate_variant:Nn \coffin_set_eq:NN { c , Nc , cc }
\cs_new_protected:Npn \coffin_gset_eq:NN #1#2
  {
    \__coffin_if_exist:NT #2
      {
        \box_gset_eq:NN #1 #2
        \prop_gset_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ corners }
          { coffin ~ \__coffin_to_value:N #2 ~ corners }
        \prop_gset_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ poles }
          { coffin ~ \__coffin_to_value:N #2 ~ poles }
      }
  }
\cs_generate_variant:Nn \coffin_gset_eq:NN { c , Nc , cc }
```

(*End of definition for* `\coffin_set_eq:NN` *and* `\coffin_gset_eq:NN`*. These functions are documented on page 314.*)

`\c_empty_coffin` `\l__coffin_aligned_coffin` `\l__coffin_aligned_internal_coffin`

Special coffins: these cannot be set up earlier as they need `\coffin_new:N`. The empty coffin is set as a box as the full coffin-setting system needs some material which is not yet available. The empty coffin is created entirely by hand: not everything is in place yet.

```
\coffin_new:N \c_empty_coffin
\coffin_new:N \l__coffin_aligned_coffin
\coffin_new:N \l__coffin_aligned_internal_coffin
```

(*End of definition for* `\c_empty_coffin` *,* `\l__coffin_aligned_coffin` *, and* `\l__coffin_aligned_internal_coffin`*. This variable is documented on page 319.*)

`\l_tmpa_coffin` `\l_tmpb_coffin` `\g_tmpa_coffin` `\g_tmpb_coffin`

The usual scratch space.

```
\coffin_new:N \l_tmpa_coffin
\coffin_new:N \l_tmpb_coffin
\coffin_new:N \g_tmpa_coffin
\coffin_new:N \g_tmpb_coffin
```

(*End of definition for* `\l_tmpa_coffin` *and others. These variables are documented on page 319.*)

92.3 Measuring coffins

`\coffin_dp:N` `\coffin_dp:c` `\coffin_ht:N` `\coffin_ht:c` `\coffin_ht_plus_dp:N` `\coffin_ht_plus_dp:c` `\coffin_wd:N` `\coffin_wd:c`

Coffins are just boxes when it comes to measurement. However, semantically a separate set of functions are required.

```
\cs_new_eq:NN \coffin_dp:N \box_dp:N
\cs_new_eq:NN \coffin_dp:c \box_dp:c
\cs_new_eq:NN \coffin_ht:N \box_ht:N
\cs_new_eq:NN \coffin_ht:c \box_ht:c
\cs_new_eq:NN \coffin_ht_plus_dp:N \box_ht_plus_dp:N
\cs_new_eq:NN \coffin_ht_plus_dp:c \box_ht_plus_dp:c
\cs_new_eq:NN \coffin_wd:N \box_wd:N
\cs_new_eq:NN \coffin_wd:c \box_wd:c
```

(*End of definition for* `\coffin_dp:N` *and others. These functions are documented on page 317.*)

92.4 Coffins: handle and pole management

`\__coffin_get_pole:NnN`

A simple wrapper around the recovery of a coffin pole, with some error checking and recovery built-in.

```
\cs_new_protected:Npn \__coffin_get_pole:NnN #1#2#3
  {
    \prop_get:cnNF
      { coffin ~ \__coffin_to_value:N #1 ~ poles } {#2} #3
      {
        \msg_error:nnee { coffin } { unknown-pole }
          { \exp_not:n {#2} } { \token_to_str:N #1 }
        \tl_set:Nn #3 { { 0pt } { 0pt } { 0pt } { 0pt } }
      }
  }
```

(*End of definition for* `\__coffin_get_pole:NnN`*.*)

`\__coffin_reset_structure:N` `\__coffin_greset_structure:N`

Resetting the structure is a simple copy job.

```
\cs_new_protected:Npn \__coffin_reset_structure:N #1
  {
    \prop_set_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
      \c__coffin_corners_prop
    \prop_set_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
```

```
      \c__coffin_poles_prop
  }
\cs_new_protected:Npn \__coffin_greset_structure:N #1
  {
    \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
      \c__coffin_corners_prop
    \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
      \c__coffin_poles_prop
  }
```

(*End of definition for* `\__coffin_reset_structure:N` *and* `\__coffin_greset_structure:N`*.*)

\coffin_set_horizontal_pole:Nnn
\coffin_set_horizontal_pole:cnn
\coffin_gset_horizontal_pole:Nnn
\coffin_gset_horizontal_pole:cnn
__coffin_set_horizontal_pole:NnnN
\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnn
\coffin_gset_vertical_pole:Nnn
\coffin_gset_vertical_pole:cnn
__coffin_set_vertical_pole:NnnN
__coffin_set_pole:Nnn
__coffin_gset_pole:Nnn

Setting the pole of a coffin at the user/designer level requires a bit more care. The idea here is to provide a reasonable interface to the system, then to do the setting with full expansion. The three-argument version is used internally to do a direct setting.

```
\cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
  { \__coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_put:cne }
\cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
\cs_new_protected:Npn \coffin_gset_horizontal_pole:Nnn #1#2#3
  { \__coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_gput:cne }
\cs_generate_variant:Nn \coffin_gset_horizontal_pole:Nnn { c }
\cs_new_protected:Npn \__coffin_set_horizontal_pole:NnnN #1#2#3#4
  {
    \__coffin_if_exist:NT #1
      {
        #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
          {#2}
          {
            { 0pt } { \dim_eval:n {#3} }
            { 1000pt } { 0pt }
          }
      }
  }
\cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
  { \__coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_put:cne }
\cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
\cs_new_protected:Npn \coffin_gset_vertical_pole:Nnn #1#2#3
  { \__coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_gput:cne }
  \cs_generate_variant:Nn \coffin_gset_vertical_pole:Nnn { c }
\cs_new_protected:Npn \__coffin_set_vertical_pole:NnnN #1#2#3#4
  {
    \__coffin_if_exist:NT #1
      {
        #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
          {#2}
          {
            { \dim_eval:n {#3} } { 0pt }
            { 0pt } { 1000pt }
          }
      }
  }
\cs_new_protected:Npn \__coffin_set_pole:Nnn #1#2#3
  {
    \prop_put:cne { coffin ~ \__coffin_to_value:N #1 ~ poles }
```

```
      {#2} {#3}
  }
\cs_new_protected:Npn \__coffin_gset_pole:Nnn #1#2#3
  {
    \prop_gput:cne { coffin ~ \__coffin_to_value:N #1 ~ poles }
      {#2} {#3}
  }
```

(*End of definition for* `\coffin_set_horizontal_pole:Nnn` *and others. These functions are documented on page 315.*)

`\coffin_reset_poles:N`
`\coffin_greset_poles:N`

Simple shortcuts.

```
\cs_new_protected:Npn \coffin_reset_poles:N #1
  {
    \__coffin_reset_structure:N #1
    \__coffin_update_corners:N #1
    \__coffin_update_poles:N #1
  }
\cs_new_protected:Npn \coffin_greset_poles:N #1
  {
    \__coffin_greset_structure:N #1
    \__coffin_gupdate_corners:N #1
    \__coffin_gupdate_poles:N #1
  }
```

(*End of definition for* `\coffin_reset_poles:N` *and* `\coffin_greset_poles:N`*. These functions are documented on page 316.*)

`\__coffin_update_corners:N`
`\__coffin_gupdate_corners:N`
`\__coffin_update_corners:NN`
`\__coffin_update_corners:NNN`

Updating the corners of a coffin is straight-forward as at this stage there can be no rotation. So the corners of the content are just those of the underlying TEX box.

```
\cs_new_protected:Npn \__coffin_update_corners:N #1
  { \__coffin_update_corners:NN #1 \prop_put:Nne }
\cs_new_protected:Npn \__coffin_gupdate_corners:N #1
  { \__coffin_update_corners:NN #1 \prop_gput:Nne }
\cs_new_protected:Npn \__coffin_update_corners:NN #1#2
  {
    \exp_args:Nc \__coffin_update_corners:NNN
      { coffin ~ \__coffin_to_value:N #1 ~ corners }
      #1 #2
  }
\cs_new_protected:Npn \__coffin_update_corners:NNN #1#2#3
  {
    #3 #1
      { tl }
      { { 0pt } { \dim_eval:n { \box_ht:N #2 } } }
    #3 #1
      { tr }
      {
        { \dim_eval:n { \box_wd:N #2 } }
        { \dim_eval:n { \box_ht:N #2 } }
      }
    #3 #1
      { bl }
      { { 0pt } { \dim_eval:n { -\box_dp:N #2 } } }
    #3 #1
```

```
      { br }
      {
        { \dim_eval:n { \box_wd:N #2 } }
        { \dim_eval:n { -\box_dp:N #2 } }
      }
  }
```

(*End of definition for* `\__coffin_update_corners:N` *and others.*)

`\__coffin_update_poles:N`
`\__coffin_gupdate_poles:N`
`\__coffin_update_poles:NN`
`\__coffin_update_poles:NNN`

This function is called when a coffin is set, and updates the poles to reflect the nature of size of the box. Thus this function only alters poles where the default position is dependent on the size of the box. It also does not set poles which are relevant only to vertical coffins.

```
\cs_new_protected:Npn \__coffin_update_poles:N #1
  { \__coffin_update_poles:NN #1 \prop_put:Nne }
\cs_new_protected:Npn \__coffin_gupdate_poles:N #1
  { \__coffin_update_poles:NN #1 \prop_gput:Nne }
\cs_new_protected:Npn \__coffin_update_poles:NN #1#2
  {
    \exp_args:Nc \__coffin_update_poles:NNN
      { coffin ~ \__coffin_to_value:N #1 ~ poles }
      #1 #2
  }
\cs_new_protected:Npn \__coffin_update_poles:NNN #1#2#3
  {
    #3 #1 { hc }
      {
        { \dim_eval:n { 0.5 \box_wd:N #2 } }
        { 0pt } { 0pt } { 1000pt }
      }
    #3 #1 { r }
      {
        { \dim_eval:n { \box_wd:N #2 } }
        { 0pt } { 0pt } { 1000pt }
      }
    #3 #1 { vc }
      {
        { 0pt }
        { \dim_eval:n { ( \box_ht:N #2 - \box_dp:N #2 ) / 2 } }
        { 1000pt }
        { 0pt }
      }
    #3 #1 { t }
      {
        { 0pt }
        { \dim_eval:n { \box_ht:N #2 } }
        { 1000pt }
        { 0pt }
      }
    #3 #1 { b }
      {
        { 0pt }
        { \dim_eval:n { -\box_dp:N #2 } }
        { 1000pt }
```

```
        { 0pt }
      }
  }
```

(*End of definition for* `\__coffin_update_poles:N` *and others.*)

92.5 Coffins: calculation of pole intersections

__coffin_calculate_intersection:Nnn
__coffin_calculate_intersection:nnnnnnnn
__coffin_calculate_intersection:nnnnnn

The lead off in finding intersections is to recover the two poles and then hand off to the auxiliary for the actual calculation. There may of course not be an intersection, for which an error trap is needed.

```
\cs_new_protected:Npn \__coffin_calculate_intersection:Nnn #1#2#3
  {
    \__coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
    \__coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
    \bool_set_false:N \l__coffin_error_bool
    \exp_last_two_unbraced:Noo
      \__coffin_calculate_intersection:nnnnnnnn
        \l__coffin_pole_a_tl \l__coffin_pole_b_tl
    \bool_if:NT \l__coffin_error_bool
      {
        \msg_error:nn { coffin } { no-pole-intersection }
        \dim_zero:N \l__coffin_x_dim
        \dim_zero:N \l__coffin_y_dim
      }
  }
```

The two poles passed here each have four values (as dimensions), (a, b, c, d) and (a', b', c', d'). These are arguments 1–4 and 5–8, respectively. In both cases a and b are the coordinates of a point on the pole and c and d define the direction of the pole. Finding the intersection depends on the directions of the poles, which are given by d/c and d'/c'. However, if one of the poles is either horizontal or vertical then one or more of c, d, c' and d' are zero and a special case is needed.

```
\cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnnnn
  #1#2#3#4#5#6#7#8
  {
    \dim_compare:nNnTF {#3} = \c_zero_dim
```

The case where the first pole is vertical. So the x-component of the interaction is at a. There is then a test on the second pole: if it is also vertical then there is an error.

```
      {
        \dim_set:Nn \l__coffin_x_dim {#1}
        \dim_compare:nNnTF {#7} = \c_zero_dim
          { \bool_set_true:N \l__coffin_error_bool }
```

The second pole may still be horizontal, in which case the y-component of the intersection is b'. If not,

$$y = \frac{d'}{c'}\,(a - a') + b'$$

with the x-component already known to be `#1`.

```
          {
            \dim_set:Nn \l__coffin_y_dim
              {
```

```
                    \dim_compare:nNnTF {#8} = \c_zero_dim
                      {#6}
                      {
                        \fp_to_dim:n
                          {
                              ( \dim_to_fp:n {#8} / \dim_to_fp:n {#7} )
                            * ( \dim_to_fp:n {#1} - \dim_to_fp:n {#5} )
                            + \dim_to_fp:n {#6}
                          }
                      }
                  }
              }
          }
```

If the first pole is not vertical then it may be horizontal. If so, then the procedure is essentially the same as that already done but with the x- and y-components interchanged.

```
          {
            \dim_compare:nNnTF {#4} = \c_zero_dim
              {
                \dim_set:Nn \l__coffin_y_dim {#2}
                \dim_compare:nNnTF {#8} = { \c_zero_dim }
                  { \bool_set_true:N \l__coffin_error_bool }
                  {
```

Now we deal with the case where the second pole may be vertical, or if not we have

$$x = \frac{c'}{d'}(b - b') + a'$$

which is again handled by the same auxiliary.

```
                    \dim_set:Nn \l__coffin_x_dim
                      {
                        \dim_compare:nNnTF {#7} = \c_zero_dim
                          {#5}
                          {
                            \fp_to_dim:n
                              {
                                  ( \dim_to_fp:n {#7} / \dim_to_fp:n {#8} )
                                * ( \dim_to_fp:n {#4} - \dim_to_fp:n {#6} )
                                + \dim_to_fp:n {#5}
                              }
                          }
                      }
                  }
              }
```

The first pole is neither horizontal nor vertical. To avoid even more complexity, we now work out both slopes and pass to an auxiliary.

```
              {
                \use:e
                  {
                    \__coffin_calculate_intersection:nnnnnn
                      { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
                      { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
                  }
                    {#1} {#2} {#5} {#6}
```

```
        }
    }
}
```

Assuming the two poles are not parallel, then the intersection point is found in two steps. First we find the x-value with

$$x = \frac{sa - s'a' - b + b'}{s - s'}$$

and then finding the y-value with

$$y = s(x - a) + b$$

```
\cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnn #1#2#3#4#5#6
  {
    \fp_compare:nNnTF {#1} = {#2}
      { \bool_set_true:N \l__coffin_error_bool }
      {
        \dim_set:Nn \l__coffin_x_dim
          {
            \fp_to_dim:n
              {
                (
                    #1 * \dim_to_fp:n {#3}
                  - #2 * \dim_to_fp:n {#5}
                  - \dim_to_fp:n {#4}
                  + \dim_to_fp:n {#6}
                )
                /
                ( #1 - #2 )
              }
          }
        \dim_set:Nn \l__coffin_y_dim
          {
            \fp_to_dim:n
              {
                  #1 * ( \l__coffin_x_dim - \dim_to_fp:n {#3} )
                + \dim_to_fp:n {#4}
              }
          }
      }
  }
```

(*End of definition for `\__coffin_calculate_intersection:Nnn`, `\__coffin_calculate_intersection:nnnnnnnn`, and `\__coffin_calculate_intersection:nnnnnn`.*)

92.6 Affine transformations

`\l__coffin_sin_fp`
`\l__coffin_cos_fp`

Used for rotations to get the sine and cosine values.

```
\fp_new:N \l__coffin_sin_fp
\fp_new:N \l__coffin_cos_fp
```

(*End of definition for `\l__coffin_sin_fp` and `\l__coffin_cos_fp`.*)

`\l__coffin_bounding_prop`

A property list for the bounding box of a coffin. This is only needed during the rotation, so there is just the one.

```
\prop_new:N \l__coffin_bounding_prop
```

(*End of definition for* `\l__coffin_bounding_prop`.)

`\l__coffin_corners_prop`
`\l__coffin_poles_prop`

Used to avoid needing to track scope for intermediate steps.

```
\prop_new:N \l__coffin_corners_prop
\prop_new:N \l__coffin_poles_prop
```

(*End of definition for* `\l__coffin_corners_prop` *and* `\l__coffin_poles_prop`.)

`\l__coffin_bounding_shift_dim`

The shift of the bounding box of a coffin from the real content.

```
\dim_new:N \l__coffin_bounding_shift_dim
```

(*End of definition for* `\l__coffin_bounding_shift_dim`.)

`\l__coffin_left_corner_dim`
`\l__coffin_right_corner_dim`
`\l__coffin_bottom_corner_dim`
`\l__coffin_top_corner_dim`

These are used to hold maxima for the various corner values: these thus define the minimum size of the bounding box after rotation.

```
\dim_new:N \l__coffin_left_corner_dim
\dim_new:N \l__coffin_right_corner_dim
\dim_new:N \l__coffin_bottom_corner_dim
\dim_new:N \l__coffin_top_corner_dim
```

(*End of definition for* `\l__coffin_left_corner_dim` *and others*.)

`\coffin_rotate:Nn`
`\coffin_rotate:cn`
`\coffin_grotate:Nn`
`\coffin_grotate:cn`
`\__coffin_rotate:NnNNN`

Rotating a coffin requires several steps which can be conveniently run together. The sine and cosine of the angle in degrees are computed. This is then used to set `\l__coffin_sin_fp` and `\l__coffin_cos_fp`, which are carried through unchanged for the rest of the procedure.

```
\cs_new_protected:Npn \coffin_rotate:Nn #1#2
  { \__coffin_rotate:NnNNN #1 {#2} \box_rotate:Nn \prop_set_eq:cN \hbox_set:Nn }
\cs_generate_variant:Nn \coffin_rotate:Nn { c }
\cs_new_protected:Npn \coffin_grotate:Nn #1#2
  { \__coffin_rotate:NnNNN #1 {#2} \box_grotate:Nn \prop_gset_eq:cN \hbox_gset:Nn }
\cs_generate_variant:Nn \coffin_grotate:Nn { c }
\cs_new_protected:Npn \__coffin_rotate:NnNNN #1#2#3#4#5
  {
    \fp_set:Nn \l__coffin_sin_fp { sind ( #2 ) }
    \fp_set:Nn \l__coffin_cos_fp { cosd ( #2 ) }
```

Use a local copy of the property lists to avoid needing to pass the name and scope around.

```
    \prop_set_eq:Nc \l__coffin_corners_prop
      { coffin ~ \__coffin_to_value:N #1 ~ corners }
    \prop_set_eq:Nc \l__coffin_poles_prop
      { coffin ~ \__coffin_to_value:N #1 ~ poles }
```

The corners and poles of the coffin can now be rotated around the origin. This is best achieved using mapping functions.

```
    \prop_map_inline:Nn \l__coffin_corners_prop
      { \__coffin_rotate_corner:Nnnn #1 {##1} ##2 }
    \prop_map_inline:Nn \l__coffin_poles_prop
      { \__coffin_rotate_pole:Nnnnnn #1 {##1} ##2 }
```

The bounding box of the coffin needs to be rotated, and to do this the corners have to be found first. They are then rotated in the same way as the corners of the coffin material itself.

```
    \__coffin_set_bounding:N #1
    \prop_map_inline:Nn \l__coffin_bounding_prop
      { \__coffin_rotate_bounding:nnn {##1} ##2 }
```

At this stage, there needs to be a calculation to find where the corners of the content and the box itself will end up.

```
    \__coffin_find_corner_maxima:N #1
    \__coffin_find_bounding_shift:
    #3 #1 {#2}
```

The correction of the box position itself takes place here. The idea is that the bounding box for a coffin is tight up to the content, and has the reference point at the bottom-left. The x-direction is handled by moving the content by the difference in the positions of the bounding box and the content left edge. The y-direction is dealt with by moving the box down by any depth it has acquired. The internal box is used here to allow for the next step.

```
    \hbox_set:Nn \l__coffin_internal_box
      {
        \__kernel_kern:n
            { \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim }
        \box_move_down:nn { \l__coffin_bottom_corner_dim }
          { \box_use:N #1 }
      }
```

If there have been any previous rotations then the size of the bounding box will be bigger than the contents. This can be corrected easily by setting the size of the box to the height and width of the content. As this operation requires setting box dimensions and these transcend grouping, the safe way to do this is to use the internal box and to reset the result into the target box.

```
    \box_set_ht:Nn \l__coffin_internal_box
      { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
    \box_set_dp:Nn \l__coffin_internal_box { 0pt }
    \box_set_wd:Nn \l__coffin_internal_box
      { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
    #5 #1 { \box_use_drop:N \l__coffin_internal_box }
```

The final task is to move the poles and corners such that they are back in alignment with the box reference point.

```
    \prop_map_inline:Nn \l__coffin_corners_prop
      { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
    \prop_map_inline:Nn \l__coffin_poles_prop
      { \__coffin_shift_pole:Nnnnnn #1 {##1} ##2 }
```

Update the coffin data.

```
    #4 { coffin ~ \__coffin_to_value:N #1 ~ corners }
      \l__coffin_corners_prop
    #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
      \l__coffin_poles_prop
  }
```

(*End of definition for* `\coffin_rotate:Nn`*,* `\coffin_grotate:Nn`*, and* `\__coffin_rotate:NnNNN`*. These functions are documented on page 316.*)

__coffin_set_bounding:N The bounding box corners for a coffin are easy enough to find: this is the same code as for the corners of the material itself, but using a dedicated property list.

```
\cs_new_protected:Npn \__coffin_set_bounding:N #1
  {
    \prop_put:Nne \l__coffin_bounding_prop { tl }
      { { 0pt } { \dim_eval:n { \box_ht:N #1 } } }
    \prop_put:Nne \l__coffin_bounding_prop { tr }
      {
        { \dim_eval:n { \box_wd:N #1 } }
        { \dim_eval:n { \box_ht:N #1 } }
      }
    \dim_set:Nn \l__coffin_internal_dim { -\box_dp:N #1 }
    \prop_put:Nne \l__coffin_bounding_prop { bl }
      { { 0pt } { \dim_use:N \l__coffin_internal_dim } }
    \prop_put:Nne \l__coffin_bounding_prop { br }
      {
        { \dim_eval:n { \box_wd:N #1 } }
        { \dim_use:N \l__coffin_internal_dim }
      }
  }
```

(*End of definition for* `\__coffin_set_bounding:N`.)

__coffin_rotate_bounding:nnn
__coffin_rotate_corner:Nnnn

Rotating the position of the corner of the coffin is just a case of treating this as a vector from the reference point. The same treatment is used for the corners of the material itself and the bounding box.

```
\cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
  {
    \__coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
    \prop_put:Nne \l__coffin_bounding_prop {#1}
      { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
  }
\cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
  {
    \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
    \prop_put:Nne \l__coffin_corners_prop {#2}
      { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
  }
```

(*End of definition for* `\__coffin_rotate_bounding:nnn` *and* `\__coffin_rotate_corner:Nnnn`.)

__coffin_rotate_pole:Nnnnnn Rotating a single pole simply means shifting the coordinate of the pole and its direction. The rotation here is about the bottom-left corner of the coffin.

```
\cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
  {
    \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
    \__coffin_rotate_vector:nnNN {#5} {#6}
      \l__coffin_x_prime_dim \l__coffin_y_prime_dim
    \prop_put:Nne \l__coffin_poles_prop {#2}
      {
        { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
        { \dim_use:N \l__coffin_x_prime_dim }
        { \dim_use:N \l__coffin_y_prime_dim }
      }
  }
```

(*End of definition for* `\__coffin_rotate_pole:Nnnnnn`.)

`\__coffin_rotate_vector:nnNN`

A rotation function, which needs only an input vector (as dimensions) and an output space. The values `\l__coffin_cos_fp` and `\l__coffin_sin_fp` should previously have been set up correctly. Working this way means that the floating point work is kept to a minimum: for any given rotation the sin and cosine values do no change, after all.

```
\cs_new_protected:Npn \__coffin_rotate_vector:nnNN #1#2#3#4
  {
    \dim_set:Nn #3
      {
        \fp_to_dim:n
          {
              \dim_to_fp:n {#1} * \l__coffin_cos_fp
            - \dim_to_fp:n {#2} * \l__coffin_sin_fp
          }
      }
    \dim_set:Nn #4
      {
        \fp_to_dim:n
          {
              \dim_to_fp:n {#1} * \l__coffin_sin_fp
            + \dim_to_fp:n {#2} * \l__coffin_cos_fp
          }
      }
  }
```

(*End of definition for* `\__coffin_rotate_vector:nnNN`.)

`\__coffin_find_corner_maxima:N`
`\__coffin_find_corner_maxima_aux:nn`

The idea here is to find the extremities of the content of the coffin. This is done by looking for the smallest values for the bottom and left corners, and the largest values for the top and right corners. The values start at the maximum dimensions so that the case where all are positive or all are negative works out correctly.

```
\cs_new_protected:Npn \__coffin_find_corner_maxima:N #1
  {
    \dim_set:Nn \l__coffin_top_corner_dim   { -\c_max_dim }
    \dim_set:Nn \l__coffin_right_corner_dim { -\c_max_dim }
    \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
    \dim_set:Nn \l__coffin_left_corner_dim   { \c_max_dim }
    \prop_map_inline:Nn \l__coffin_corners_prop
      { \__coffin_find_corner_maxima_aux:nn ##2 }
  }
\cs_new_protected:Npn \__coffin_find_corner_maxima_aux:nn #1#2
  {
    \dim_set:Nn \l__coffin_left_corner_dim
      { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
    \dim_set:Nn \l__coffin_right_corner_dim
      { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
    \dim_set:Nn \l__coffin_bottom_corner_dim
      { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
    \dim_set:Nn \l__coffin_top_corner_dim
      { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
  }
```

(*End of definition for* `\__coffin_find_corner_maxima:N` *and* `\__coffin_find_corner_maxima_aux:nn`.)

__coffin_find_bounding_shift:
__coffin_find_bounding_shift_aux:nn

The approach to finding the shift for the bounding box is similar to that for the corners. However, there is only one value needed here and a fixed input property list, so things are a bit clearer.

```
\cs_new_protected:Npn \__coffin_find_bounding_shift:
  {
    \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }
    \prop_map_inline:Nn \l__coffin_bounding_prop
      { \__coffin_find_bounding_shift_aux:nn ##2 }
  }
\cs_new_protected:Npn \__coffin_find_bounding_shift_aux:nn #1#2
  {
    \dim_set:Nn \l__coffin_bounding_shift_dim
      { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
  }
```

(*End of definition for* `\__coffin_find_bounding_shift:` *and* `\__coffin_find_bounding_shift_aux:nn`.)

__coffin_shift_corner:Nnnn
__coffin_shift_pole:Nnnnnn

Shifting the corners and poles of a coffin means subtracting the appropriate values from the x- and y-components. For the poles, this means that the direction vector is unchanged.

```
\cs_new_protected:Npn \__coffin_shift_corner:Nnnn #1#2#3#4
  {
    \prop_put:Nne \l__coffin_corners_prop {#2}
      {
        { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
        { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
      }
  }
\cs_new_protected:Npn \__coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
  {
    \prop_put:Nne \l__coffin_poles_prop {#2}
      {
        { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
        { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
        {#5} {#6}
      }
  }
```

(*End of definition for* `\__coffin_shift_corner:Nnnn` *and* `\__coffin_shift_pole:Nnnnnn`.)

\l__coffin_scale_x_fp
\l__coffin_scale_y_fp

Storage for the scaling factors in x and y, respectively.

```
\fp_new:N \l__coffin_scale_x_fp
\fp_new:N \l__coffin_scale_y_fp
```

(*End of definition for* `\l__coffin_scale_x_fp` *and* `\l__coffin_scale_y_fp`.)

\l__coffin_scaled_total_height_dim
\l__coffin_scaled_width_dim

When scaling, the values given have to be turned into absolute values.

```
\dim_new:N \l__coffin_scaled_total_height_dim
\dim_new:N \l__coffin_scaled_width_dim
```

(*End of definition for* `\l__coffin_scaled_total_height_dim` *and* `\l__coffin_scaled_width_dim`.)

\coffin_resize:Nnn
\coffin_resize:cnn
\coffin_gresize:Nnn
\coffin_gresize:cnn
__coffin_resize:NnnNN

Resizing a coffin begins by setting up the user-friendly names for the dimensions of the coffin box. The new sizes are then turned into scale factor. This is the same operation as takes place for the underlying box, but that operation is grouped and so the same calculation is done here.

```
\cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
  {
    \__coffin_resize:NnnNN #1 {#2} {#3}
      \box_resize_to_wd_and_ht_plus_dp:Nnn
      \prop_set_eq:cN
  }
\cs_generate_variant:Nn \coffin_resize:Nnn { c }
\cs_new_protected:Npn \coffin_gresize:Nnn #1#2#3
  {
    \__coffin_resize:NnnNN #1 {#2} {#3}
      \box_gresize_to_wd_and_ht_plus_dp:Nnn
      \prop_gset_eq:cN
  }
\cs_generate_variant:Nn \coffin_gresize:Nnn { c }
\cs_new_protected:Npn \__coffin_resize:NnnNN #1#2#3#4#5
  {
    \fp_set:Nn \l__coffin_scale_x_fp
      { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
    \fp_set:Nn \l__coffin_scale_y_fp
      {
          \dim_to_fp:n {#3}
        / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
      }
    #4 #1 {#2} {#3}
    \__coffin_resize_common:NnnN #1 {#2} {#3} #5
  }
```

(*End of definition for* `\coffin_resize:Nnn` , `\coffin_gresize:Nnn` , *and* `\__coffin_resize:NnnNN`*. These functions are documented on page 316.*)

__coffin_resize_common:NnnN

The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```
\cs_new_protected:Npn \__coffin_resize_common:NnnN #1#2#3#4
  {
    \prop_set_eq:Nc \l__coffin_corners_prop
      { coffin ~ \__coffin_to_value:N #1 ~ corners }
    \prop_set_eq:Nc \l__coffin_poles_prop
      { coffin ~ \__coffin_to_value:N #1 ~ poles }
    \prop_map_inline:Nn \l__coffin_corners_prop
      { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }
    \prop_map_inline:Nn \l__coffin_poles_prop
      { \__coffin_scale_pole:Nnnnnn #1 {##1} ##2 }
```

Negative x-scaling values place the poles in the wrong location: this is corrected here.

```
    \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
      {
        \prop_map_inline:Nn \l__coffin_corners_prop
          { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
        \prop_map_inline:Nn \l__coffin_poles_prop
          { \__coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
```

```
      }
    #4 { coffin ~ \__coffin_to_value:N #1 ~ corners }
      \l__coffin_corners_prop
    #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
      \l__coffin_poles_prop
  }
```

(*End of definition for* `\__coffin_resize_common:NnnN`.)

`\coffin_scale:Nnn`
`\coffin_scale:cnn`
`\coffin_gscale:Nnn`
`\coffin_gscale:cnn`
`\__coffin_scale:NnnNN`

For scaling, the opposite calculation is done to find the new dimensions for the coffin. Only the total height is needed, as this is the shift required for corners and poles. The scaling is done the TeX way as this works properly with floating point values without needing to use the `fp` module.

```
\cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
  { \__coffin_scale:NnnNN #1 {#2} {#3} \box_scale:Nnn \prop_set_eq:cN }
\cs_generate_variant:Nn \coffin_scale:Nnn { c }
\cs_new_protected:Npn \coffin_gscale:Nnn #1#2#3
  { \__coffin_scale:NnnNN #1 {#2} {#3} \box_gscale:Nnn \prop_gset_eq:cN }
\cs_generate_variant:Nn \coffin_gscale:Nnn { c }
\cs_new_protected:Npn \__coffin_scale:NnnNN #1#2#3#4#5
  {
    \fp_set:Nn \l__coffin_scale_x_fp {#2}
    \fp_set:Nn \l__coffin_scale_y_fp {#3}
    #4 #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }
    \dim_set:Nn \l__coffin_internal_dim
      { \coffin_ht:N #1 + \coffin_dp:N #1 }
    \dim_set:Nn \l__coffin_scaled_total_height_dim
      { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_internal_dim }
    \dim_set:Nn \l__coffin_scaled_width_dim
      { -\fp_abs:n { \l__coffin_scale_x_fp  } \coffin_wd:N #1 }
    \__coffin_resize_common:NnnN #1
      { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
      #5
  }
```

(*End of definition for* `\coffin_scale:Nnn`, `\coffin_gscale:Nnn`, *and* `\__coffin_scale:NnnNN`. *These functions are documented on page 316.*)

`\__coffin_scale_vector:nnNN`

This functions scales a vector from the origin using the pre-set scale factors in x and y. This is a much less complex operation than rotation, and as a result the code is a lot clearer.

```
\cs_new_protected:Npn \__coffin_scale_vector:nnNN #1#2#3#4
  {
    \dim_set:Nn #3
      { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
    \dim_set:Nn #4
      { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
  }
```

(*End of definition for* `\__coffin_scale_vector:nnNN`.)

`\__coffin_scale_corner:Nnnn`
`\__coffin_scale_pole:Nnnnnn`

Scaling both corners and poles is a simple calculation using the preceding vector scaling.

```
\cs_new_protected:Npn \__coffin_scale_corner:Nnnn #1#2#3#4
  {
```

```
    \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
    \prop_put:Nne \l__coffin_corners_prop {#2}
      { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
  }
\cs_new_protected:Npn \__coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
  {
    \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
    \prop_put:Nne \l__coffin_poles_prop {#2}
      {
        { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
        {#5} {#6}
      }
  }
```

(*End of definition for* `\__coffin_scale_corner:Nnnn` *and* `\__coffin_scale_pole:Nnnnnn`.)

`\__coffin_x_shift_corner:Nnnn`
`\__coffin_x_shift_pole:Nnnnnn`

These functions correct for the x displacement that takes place with a negative horizontal scaling.

```
\cs_new_protected:Npn \__coffin_x_shift_corner:Nnnn #1#2#3#4
  {
    \prop_put:Nne \l__coffin_corners_prop {#2}
      {
        { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
      }
  }
\cs_new_protected:Npn \__coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
  {
    \prop_put:Nne \l__coffin_poles_prop {#2}
      {
        { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
        {#5} {#6}
      }
  }
```

(*End of definition for* `\__coffin_x_shift_corner:Nnnn` *and* `\__coffin_x_shift_pole:Nnnnnn`.)

92.7 Aligning and typesetting of coffins

`\coffin_join:NnnNnnnn`
`\coffin_join:cnnNnnnn`
`\coffin_join:Nnncnnnn`
`\coffin_join:cnncnnnn`
`\coffin_gjoin:NnnNnnnn`
`\coffin_gjoin:cnnNnnnn`
`\coffin_gjoin:Nnncnnnn`
`\coffin_gjoin:cnncnnnn`
`\__coffin_join:NnnNnnnnN`

This command joins two coffins, using a horizontal and vertical pole from each coffin and making an offset between the two. The result is stored as the as a third coffin, which has all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

```
\cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
  {
    \__coffin_join:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
      \coffin_set_eq:NN
  }
\cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cnnc }
\cs_new_protected:Npn \coffin_gjoin:NnnNnnnn #1#2#3#4#5#6#7#8
  {
    \__coffin_join:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
      \coffin_gset_eq:NN
  }
```

```
\cs_generate_variant:Nn \coffin_gjoin:NnnNnnnn { c , Nnnc , cnnc }
\cs_new_protected:Npn \__coffin_join:NnnNnnnnN #1#2#3#4#5#6#7#8#9
  {
    \__coffin_align:NnnNnnnnN
      #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
```

Correct the placement of the reference point. If the x-offset is negative then the reference point of the second box is to the left of that of the first, which is corrected using a kern. On the right side the first box might stick out, which would show up if it is wider than the sum of the x-offset and the width of the second box. So a second kern may be needed.

```
    \hbox_set:Nn \l__coffin_aligned_coffin
      {
        \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
          { \__kernel_kern:n { -\l__coffin_offset_x_dim } }
        \hbox_unpack:N \l__coffin_aligned_coffin
        \dim_set:Nn \l__coffin_internal_dim
          { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
        \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
          { \__kernel_kern:n { -\l__coffin_internal_dim } }
      }
```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```
    \__coffin_reset_structure:N \l__coffin_aligned_coffin
    \prop_clear:c
      {
        coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
        \c_space_tl corners
      }
    \__coffin_update_poles:N \l__coffin_aligned_coffin
```

The structures of the parent coffins are now transferred to the new coffin, which requires that the appropriate offsets are applied. That then depends on whether any shift was needed.

```
    \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
      {
        \__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { 0pt }
        \__coffin_offset_poles:Nnn #4 { 0pt } { \l__coffin_offset_y_dim }
        \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { 0pt }
        \__coffin_offset_corners:Nnn #4 { 0pt } { \l__coffin_offset_y_dim }
      }
      {
        \__coffin_offset_poles:Nnn #1 { 0pt } { 0pt }
        \__coffin_offset_poles:Nnn #4
          { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
        \__coffin_offset_corners:Nnn #1 { 0pt } { 0pt }
        \__coffin_offset_corners:Nnn #4
          { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
      }
    \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
    #9 #1 \l__coffin_aligned_coffin
  }
```

(*End of definition for* `\coffin_join:NnnNnnnn`*,* `\coffin_gjoin:NnnNnnnn`*, and* `\__coffin_join:NnnNnnnnN`*. These functions are documented on page 317.*)

\coffin_attach:NnnNnnnn
\coffin_attach:cnnNnnnn
\coffin_attach:Nnncnnnn
\coffin_attach:cnncnnnn
\coffin_gattach:NnnNnnnn
\coffin_gattach:cnnNnnnn
\coffin_gattach:Nnncnnnn
\coffin_gattach:cnncnnnn
__coffin_attach:NnnNnnnnN
__coffin_attach_mark:NnnNnnnn

A more simple version of the above, as it simply uses the size of the first coffin for the new one. This means that the work here is rather simplified compared to the above code. The function used when marking a position is hear also as it is similar but without the structure updates.

```
\cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
  {
    \__coffin_attach:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
      \coffin_set_eq:NN
  }
\cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnnc }
\cs_new_protected:Npn \coffin_gattach:NnnNnnnn #1#2#3#4#5#6#7#8
  {
    \__coffin_attach:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
      \coffin_gset_eq:NN
  }
\cs_generate_variant:Nn \coffin_gattach:NnnNnnnn { c , Nnnc , cnnc }
\cs_new_protected:Npn \__coffin_attach:NnnNnnnnN #1#2#3#4#5#6#7#8#9
  {
    \__coffin_align:NnnNnnnnN
      #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
    \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
    \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
    \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
    \__coffin_reset_structure:N \l__coffin_aligned_coffin
    \prop_set_eq:cc
      {
        coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
        \c_space_tl corners
      }
      { coffin ~ \__coffin_to_value:N #1 ~ corners }
    \__coffin_update_poles:N  \l__coffin_aligned_coffin
    \__coffin_offset_poles:Nnn #1 { 0pt } { 0pt }
    \__coffin_offset_poles:Nnn #4
      { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
    \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
    #9 #1 \l__coffin_aligned_coffin
  }
\cs_new_protected:Npn \__coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
  {
    \__coffin_align:NnnNnnnnN
      #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
    \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
    \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
    \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
    \box_set_eq:NN #1 \l__coffin_aligned_coffin
  }
```

(*End of definition for* `\coffin_attach:NnnNnnnn` *and others. These functions are documented on page 317.*)

__coffin_align:NnnNnnnnN

The internal function aligns the two coffins into a third one, but performs no corrections on the resulting coffin poles. The process begins by finding the points of intersection for the poles for each of the input coffins. Those for the first coffin are worked out after those for the second coffin, as this allows the ‘primed’ storage area to be used for the second coffin. The 'real' box offsets are then calculated, before using these to re-box the input coffins. The default poles are then set up, but the final result depends on how the bounding box is being handled.

```
\cs_new_protected:Npn \__coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
  {
    \__coffin_calculate_intersection:Nnn #4 {#5} {#6}
    \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
    \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
    \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
    \dim_set:Nn \l__coffin_offset_x_dim
      { \l__coffin_x_dim - \l__coffin_x_prime_dim + #7 }
    \dim_set:Nn \l__coffin_offset_y_dim
      { \l__coffin_y_dim - \l__coffin_y_prime_dim + #8 }
    \hbox_set:Nn \l__coffin_aligned_internal_coffin
      {
        \box_use:N #1
        \__kernel_kern:n { -\box_wd:N #1 }
        \__kernel_kern:n { \l__coffin_offset_x_dim }
        \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #4 }
      }
    \coffin_set_eq:NN #9 \l__coffin_aligned_internal_coffin
  }
```

(*End of definition for* `\__coffin_align:NnnNnnnnN`.)

`\__coffin_offset_poles:Nnn`
`\__coffin_offset_pole:Nnnnnnn`

Transferring structures from one coffin to another requires that the positions are updated by the offset between the two coffins. This is done by mapping over the property list of the source coffins, moving as appropriate and saving to the new coffin data structures. The test for a `-` means that the structures from the parent coffins are uniquely labelled and do not depend on the order of alignment. The pay off for this is that `-` should not be used in coffin pole or handle names, and that multiple alignments do not result in a whole set of values.

```
\cs_new_protected:Npn \__coffin_offset_poles:Nnn #1#2#3
  {
    \prop_map_inline:cn { coffin ~ \__coffin_to_value:N #1 ~ poles }
      { \__coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {#2} {#3} }
  }
\cs_new_protected:Npn \__coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
  {
    \dim_set:Nn \l__coffin_x_dim { #3 + #7 }
    \dim_set:Nn \l__coffin_y_dim { #4 + #8 }
    \tl_if_in:nnTF {#2} { - }
      { \tl_set:Nn \l__coffin_internal_tl { {#2} } }
      { \tl_set:Nn \l__coffin_internal_tl { { #1 - #2 } } }
    \exp_last_unbraced:NNo \__coffin_set_pole:Nnn \l__coffin_aligned_coffin
      { \l__coffin_internal_tl }
      {
        { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
        {#5} {#6}
      }
  }
```

(*End of definition for* `\__coffin_offset_poles:Nnn` *and* `\__coffin_offset_pole:Nnnnnnn`.)

__coffin_offset_corners:Nnn
__coffin_offset_corner:Nnnnn

Saving the offset corners of a coffin is very similar, except that there is no need to worry about naming: every corner can be saved here as order is unimportant.

```
\cs_new_protected:Npn \__coffin_offset_corners:Nnn #1#2#3
  {
    \prop_map_inline:cn { coffin ~ \__coffin_to_value:N #1 ~ corners }
      { \__coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
  }
\cs_new_protected:Npn \__coffin_offset_corner:Nnnnn #1#2#3#4#5#6
  {
    \prop_put:cne
      {
        coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
        \c_space_tl corners
      }
      { #1 - #2 }
      {
        { \dim_eval:n { #3 + #5 } }
        { \dim_eval:n { #4 + #6 } }
      }
  }
```

(*End of definition for* `\__coffin_offset_corners:Nnn` *and* `\__coffin_offset_corner:Nnnnn`.)

__coffin_update_vertical_poles:NNN
__coffin_update_T:nnnnnnnnN
__coffin_update_B:nnnnnnnnN

The T and B poles need to be recalculated after alignment. These functions find the larger absolute value for the poles, but this is of course only logical when the poles are horizontal.

```
\cs_new_protected:Npn \__coffin_update_vertical_poles:NNN #1#2#3
  {
    \__coffin_get_pole:NnN #3 { #1 -T } \l__coffin_pole_a_tl
    \__coffin_get_pole:NnN #3 { #2 -T } \l__coffin_pole_b_tl
    \exp_last_two_unbraced:Noo \__coffin_update_T:nnnnnnnnN
      \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
    \__coffin_get_pole:NnN #3 { #1 -B } \l__coffin_pole_a_tl
    \__coffin_get_pole:NnN #3 { #2 -B } \l__coffin_pole_b_tl
    \exp_last_two_unbraced:Noo \__coffin_update_B:nnnnnnnnN
      \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
  }
\cs_new_protected:Npn \__coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
  {
    \dim_compare:nNnTF {#2} < {#6}
      {
        \__coffin_set_pole:Nnn #9 { T }
          { { 0pt } {#6} { 1000pt } { 0pt } }
      }
      {
        \__coffin_set_pole:Nnn #9 { T }
          { { 0pt } {#2} { 1000pt } { 0pt } }
      }
  }
\cs_new_protected:Npn \__coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
  {
    \dim_compare:nNnTF {#2} < {#6}
      {
        \__coffin_set_pole:Nnn #9 { B }
```

```
        { { 0pt } {#2}  { 1000pt } { 0pt } }
    }
    {
      \__coffin_set_pole:Nnn #9 { B }
        { { 0pt } {#6} { 1000pt } { 0pt } }
    }
}
```

(*End of definition for* `\__coffin_update_vertical_poles:NNN`*,* `\__coffin_update_T:nnnnnnnnN`*, and* `\__coffin_update_B:nnnnnnnnN`*.*)

`\c__coffin_empty_coffin` An empty-but-horizontal coffin.

```
\coffin_new:N \c__coffin_empty_coffin
\tex_setbox:D \c__coffin_empty_coffin = \tex_hbox:D { }
```

(*End of definition for* `\c__coffin_empty_coffin`*.*)

`\coffin_typeset:Nnnnn`
`\coffin_typeset:cnnnn`

Typesetting a coffin means aligning it with the current position, which is done using a coffin with no content at all. As well as aligning to the empty coffin, there is also a need to leave vertical mode, if necessary.

```
\cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
  {
    \mode_leave_vertical:
    \__coffin_align:NnnNnnnnN \c__coffin_empty_coffin { H } { l }
      #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
    \box_use_drop:N \l__coffin_aligned_coffin
  }
\cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }
```

(*End of definition for* `\coffin_typeset:Nnnnn`*. This function is documented on page 317.*)

92.8 Coffin diagnostics

`\l__coffin_display_coffin`
`\l__coffin_display_coord_coffin`
`\l__coffin_display_pole_coffin`

Used for printing coffins with data structures attached.

```
\coffin_new:N \l__coffin_display_coffin
\coffin_new:N \l__coffin_display_coord_coffin
\coffin_new:N \l__coffin_display_pole_coffin
```

(*End of definition for* `\l__coffin_display_coffin`*,* `\l__coffin_display_coord_coffin`*, and* `\l__coffin_display_pole_coffin`*.*)

`\l__coffin_display_handles_prop` This property list is used to print coffin handles at suitable positions. The offsets are expressed as multiples of the basic offset value, which therefore acts as a scale-factor.

```
\prop_new:N \l__coffin_display_handles_prop
\prop_put:Nnn \l__coffin_display_handles_prop { tl }
  { { b } { r } { -1 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { thc }
  { { b } { hc } { 0 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { tr }
  { { b } { l } { 1 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { vcl }
  { { vc } { r } { -1 } { 0 } }
\prop_put:Nnn \l__coffin_display_handles_prop { vchc }
  { { vc } { hc } { 0 } { 0 } }
```

```
\prop_put:Nnn \l__coffin_display_handles_prop { vcr }
  { { vc } { l } { 1 } { 0 } }
\prop_put:Nnn \l__coffin_display_handles_prop { bl }
  { { t } { r } { -1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { bhc }
  { { t } { hc } { 0 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { br }
  { { t } { l } { 1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Tl }
  { { t } { r } { -1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Thc }
  { { t } { hc } { 0 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Tr }
  { { t } { l } { 1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Hl }
  { { vc } { r } { -1 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
  { { vc } { hc } { 0 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Hr }
  { { vc } { l } { 1 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Bl }
  { { b } { r } { -1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Bhc }
  { { b } { hc } { 0 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Br }
  { { b } { l } { 1 } { -1 } }
```

(*End of definition for* `\l__coffin_display_handles_prop`.)

`\l__coffin_display_offset_dim` The standard offset for the label from the handle position when displaying handles.

```
\dim_new:N  \l__coffin_display_offset_dim
\dim_set:Nn \l__coffin_display_offset_dim { 2pt }
```

(*End of definition for* `\l__coffin_display_offset_dim`.)

`\l__coffin_display_x_dim` `\l__coffin_display_y_dim` As the intersections of poles have to be calculated to find which ones to print, there is a need to avoid repetition. This is done by saving the intersection into two dedicated values.

```
\dim_new:N \l__coffin_display_x_dim
\dim_new:N \l__coffin_display_y_dim
```

(*End of definition for* `\l__coffin_display_x_dim` *and* `\l__coffin_display_y_dim`.)

`\l__coffin_display_poles_prop` A property list for printing poles: various things need to be deleted from this to get a "nice" output.

```
\prop_new:N \l__coffin_display_poles_prop
```

(*End of definition for* `\l__coffin_display_poles_prop`.)

`\l__coffin_display_font_tl` Stores the settings used to print coffin data: this keeps things flexible.

```
\tl_new:N \l__coffin_display_font_tl
\bool_lazy_and:nnT
  { \cs_if_exist_p:N \fmtname }
  { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
  {
```

```
    \tl_set:Nn \l__coffin_display_font_tl
      { \sffamily \tiny }
  }
```

(*End of definition for* `\l__coffin_display_font_tl`.)

`\__coffin_rule:nn`

Abstract out creation of rules here until there is a higher-level interface.

```
\cs_new_protected:Npn \__coffin_rule:nn #1#2
  {
    \mode_leave_vertical:
    \hbox:n { \tex_vrule:D width #1 height #2 \scan_stop: }
  }
```

(*End of definition for* `\__coffin_rule:nn`.)

`\coffin_mark_handle:Nnnn`
`\coffin_mark_handle:cnnn`
`\__coffin_mark_handle_aux:nnnnNnn`

Marking a single handle is relatively easy. The standard attachment function is used, meaning that there are two calculations for the location. However, this is likely to be okay given the load expected. Contrast with the more optimised version for showing all handles which comes next.

```
\cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
  {
    \hcoffin_set:Nn \l__coffin_display_pole_coffin
      {
        \color_select:n {#4}
        \__coffin_rule:nn { 1pt } { 1pt }
      }
    \__coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
      \l__coffin_display_pole_coffin { hc } { vc } { 0pt } { 0pt }
    \hcoffin_set:Nn \l__coffin_display_coord_coffin
      {
        \color_select:n {#4}
        \l__coffin_display_font_tl
        ( \tl_to_str:n { #2 , #3 } )
      }
    \prop_get:NnN \l__coffin_display_handles_prop
      { #2 #3 } \l__coffin_internal_tl
    \quark_if_no_value:NTF \l__coffin_internal_tl
      {
        \prop_get:NnN \l__coffin_display_handles_prop
          { #3 #2 } \l__coffin_internal_tl
        \quark_if_no_value:NTF \l__coffin_internal_tl
          {
            \__coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
              \l__coffin_display_coord_coffin { l } { vc }
                { 1pt } { 0pt }
          }
          {
            \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
              \l__coffin_internal_tl #1 {#2} {#3}
          }
      }
      {
        \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
          \l__coffin_internal_tl #1 {#2} {#3}
```

```
      }
  }
\cs_new_protected:Npn \__coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
  {
    \__coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
      \l__coffin_display_coord_coffin {#1} {#2}
      { #3 \l__coffin_display_offset_dim }
      { #4 \l__coffin_display_offset_dim }
  }
\cs_generate_variant:Nn \coffin_mark_handle:Nnnn { c }
```

(*End of definition for* `\coffin_mark_handle:Nnnn` *and* `\__coffin_mark_handle_aux:nnnnNnn`*. This function is documented on page 318.*)

`\coffin_display_handles:Nn`
`\coffin_display_handles:cn`
`\__coffin_display_handles_aux:nnnnnn`
`\__coffin_display_handles_aux:nnnn`
`\__coffin_display_attach:Nnnnn`

Printing the poles starts by removing any duplicates, for which the `H` poles is used as the definitive version for the baseline and bottom. Two loops are then used to find the combinations of handles for all of these poles. This is done such that poles are removed during the loops to avoid duplication.

```
\cs_new_protected:Npn \coffin_display_handles:Nn #1#2
  {
    \hcoffin_set:Nn \l__coffin_display_pole_coffin
      {
        \color_select:n {#2}
        \__coffin_rule:nn { 1pt } { 1pt }
      }
    \prop_set_eq:Nc \l__coffin_display_poles_prop
      { coffin ~ \__coffin_to_value:N #1 ~ poles }
    \__coffin_get_pole:NnN #1 { H } \l__coffin_pole_a_tl
    \__coffin_get_pole:NnN #1 { T } \l__coffin_pole_b_tl
    \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
      { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
    \__coffin_get_pole:NnN #1 { B } \l__coffin_pole_b_tl
    \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
      { \prop_remove:Nn \l__coffin_display_poles_prop { B } }
    \coffin_set_eq:NN \l__coffin_display_coffin #1
    \prop_map_inline:Nn \l__coffin_display_poles_prop
      {
        \prop_remove:Nn \l__coffin_display_poles_prop {##1}
        \__coffin_display_handles_aux:nnnnnn {##1} ##2 {#2}
      }
    \box_use_drop:N \l__coffin_display_coffin
  }
```

For each pole there is a check for an intersection, which here does not give an error if none is found. The successful values are stored and used to align the pole coffin with the main coffin for output. The positions are recovered from the preset list if available.

```
\cs_new_protected:Npn \__coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
  {
    \prop_map_inline:Nn \l__coffin_display_poles_prop
      {
        \bool_set_false:N \l__coffin_error_bool
        \__coffin_calculate_intersection:nnnnnnnn {#2} {#3} {#4} {#5} ##2
        \bool_if:NF \l__coffin_error_bool
          {
```

```
            \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
            \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
            \__coffin_display_attach:Nnnnn
              \l__coffin_display_pole_coffin { hc } { vc }
              { 0pt } { 0pt }
            \hcoffin_set:Nn \l__coffin_display_coord_coffin
              {
                \color_select:n {#6}
                \l__coffin_display_font_tl
                ( \tl_to_str:n { #1 , ##1 } )
              }
            \prop_get:NnN \l__coffin_display_handles_prop
              { #1 ##1 } \l__coffin_internal_tl
            \quark_if_no_value:NTF \l__coffin_internal_tl
              {
                \prop_get:NnN \l__coffin_display_handles_prop
                  { ##1 #1 } \l__coffin_internal_tl
                \quark_if_no_value:NTF \l__coffin_internal_tl
                  {
                    \__coffin_display_attach:Nnnnn
                      \l__coffin_display_coord_coffin { l } { vc }
                      { 1pt } { 0pt }
                  }
                  {
                    \exp_last_unbraced:No
                      \__coffin_display_handles_aux:nnnn
                      \l__coffin_internal_tl
                  }
              }
              {
                \exp_last_unbraced:No \__coffin_display_handles_aux:nnnn
                  \l__coffin_internal_tl
              }
          }
      }
  }
\cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
  {
    \__coffin_display_attach:Nnnnn
      \l__coffin_display_coord_coffin {#1} {#2}
      { #3 \l__coffin_display_offset_dim }
      { #4 \l__coffin_display_offset_dim }
  }
\cs_generate_variant:Nn \coffin_display_handles:Nn { c }
```

This is a dedicated version of `\coffin_attach:NnnNnnnn` with a hard-wired first coffin. As the intersection is already known and stored for the display coffin the code simply uses it directly, with no calculation.

```
\cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5
  {
    \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
    \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
    \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
    \dim_set:Nn \l__coffin_offset_x_dim
```

```
      { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
    \dim_set:Nn \l__coffin_offset_y_dim
      { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
    \hbox_set:Nn \l__coffin_aligned_coffin
      {
        \box_use:N \l__coffin_display_coffin
        \__kernel_kern:n { -\box_wd:N \l__coffin_display_coffin }
        \__kernel_kern:n { \l__coffin_offset_x_dim }
        \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
      }
    \box_set_ht:Nn \l__coffin_aligned_coffin
      { \box_ht:N \l__coffin_display_coffin }
    \box_set_dp:Nn \l__coffin_aligned_coffin
      { \box_dp:N \l__coffin_display_coffin }
    \box_set_wd:Nn \l__coffin_aligned_coffin
      { \box_wd:N \l__coffin_display_coffin }
    \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
  }
```

(*End of definition for* `\coffin_display_handles:Nn` *and others. This function is documented on page* *318*.)

`\coffin_show_structure:N`
`\coffin_show_structure:c`
`\coffin_log_structure:N`
`\coffin_log_structure:c`
`\__coffin_show_structure:NN`

For showing the various internal structures attached to a coffin in a way that keeps things relatively readable. If there is no apparent structure then the code complains.

```
\cs_new_protected:Npn \coffin_show_structure:N
  { \__coffin_show_structure:NN \msg_show:nneeee }
\cs_generate_variant:Nn \coffin_show_structure:N { c }
\cs_new_protected:Npn \coffin_log_structure:N
  { \__coffin_show_structure:NN \msg_log:nneeee }
\cs_generate_variant:Nn \coffin_log_structure:N { c }
\cs_new_protected:Npn \__coffin_show_structure:NN #1#2
  {
    \__coffin_if_exist:NT #2
      {
        #1 { coffin } { show }
          { \token_to_str:N #2 }
          {
            \iow_newline: >~ ht ~=~ \dim_eval:n { \coffin_ht:N #2 }
            \iow_newline: >~ dp ~=~ \dim_eval:n { \coffin_dp:N #2 }
            \iow_newline: >~ wd ~=~ \dim_eval:n { \coffin_wd:N #2 }
          }
          {
            \prop_map_function:cN
              { coffin ~ \__coffin_to_value:N #2 ~ poles }
              \msg_show_item_unbraced:nn
          }
          { }
      }
  }
```

(*End of definition for* `\coffin_show_structure:N`*,* `\coffin_log_structure:N`*, and* `\__coffin_show_structure:NN`*. These functions are documented on page* *318*.)

`\coffin_show:N`
`\coffin_show:c`
`\coffin_log:N`
`\coffin_log:c`
`\coffin_show:Nnn`
`\coffin_show:cnn`
`\coffin_log:Nnn`
`\coffin_log:cnn`
`\__coffin_show:NNNnn`

Essentially a combination of `\coffin_show_structure:N` and `\box_show:Nnn`, but we need to avoid having two prompts, so we use `\msg_term:nneeee` instead of `\msg_show:nneeee` in the `show` case.

```
\cs_new_protected:Npn \coffin_show:N #1
  { \coffin_show:Nnn #1 \c_max_int \c_max_int }
\cs_generate_variant:Nn \coffin_show:N { c }
\cs_new_protected:Npn \coffin_log:N #1
  { \coffin_log:Nnn #1 \c_max_int \c_max_int }
\cs_generate_variant:Nn \coffin_log:N { c }
\cs_new_protected:Npn \coffin_show:Nnn
  { \__coffin_show:NNNnn \msg_term:nneeee \box_show:Nnn }
\cs_generate_variant:Nn \coffin_show:Nnn { c }
\cs_new_protected:Npn \coffin_log:Nnn
  { \__coffin_show:NNNnn \msg_log:nneeee \box_show:Nnn }
\cs_generate_variant:Nn \coffin_log:Nnn { c }
\cs_new_protected:Npn \__coffin_show:NNNnn #1#2#3#4#5
  {
    \__coffin_if_exist:NT #3
      {
        \__coffin_show_structure:NN #1 #3
        #2 #3 {#4} {#5}
      }
  }
```

(*End of definition for* `\coffin_show:N` *and others. These functions are documented on page 318.*)

92.9 Messages

```
\msg_new:nnnn { coffin } { no-pole-intersection }
  { No~intersection~between~coffin~poles. }
  {
    LaTeX~was~asked~to~find~the~intersection~between~two~poles,~
    but~they~do~not~have~a~unique~meeting~point:~
    the~value~(0pt,~0pt)~will~be~used.
  }
\msg_new:nnnn { coffin } { unknown }
  { Unknown~coffin~'#1'. }
  { The~coffin~'#1'~was~never~defined. }
\msg_new:nnnn { coffin } { unknown-pole }
  { Pole~'#1'~unknown~for~coffin~'#2'. }
  {
    LaTeX~was~asked~to~find~a~typesetting~pole~for~a~coffin,~
    but~either~the~coffin~does~not~exist~or~the~pole~name~is~wrong.
  }
\msg_new:nnn { coffin } { show }
  {
    Size~of~coffin~#1 : #2 \\
    Poles~of~coffin~#1 : #3 .
  }
</package>
```

Chapter 93

l3color implementation

```
⟨*package⟩
⟨@@=color⟩
```

93.1 Basics

\l__color_current_tl The color currently active for foreground (text, *etc.*) material. This is stored in the form of a color model followed by one or more values. There are four pre-defined models, three of which take numerical values in the range $[0, 1]$:

- gray ⟨*gray*⟩ Grayscale color with the ⟨*gray*⟩ value running from 0 (fully black) to 1 (fully white)
- cmyk ⟨*cyan*⟩ ⟨*magenta*⟩ ⟨*yellow*⟩ ⟨*black*⟩
- rgb ⟨*red*⟩ ⟨*green*⟩ ⟨*blue*⟩

Notice that the value are separated by spaces. There is a fourth pre-defined model using a string value and a numerical one:

- spot ⟨*name*⟩ ⟨*tint*⟩ A pre-defined spot color, where the ⟨*name*⟩ should be a pre-defined string color name and the ⟨*tint*⟩ should be in the range $[0, 1]$.

Additional models may be created to allow mixing of spot colors. The number of data entries these require will depend on the number of colors to be mixed.

TEXhackers note: The content of \l__color_current_tl comprises two brace groups, the first containing the color model and the second containing the value(s) applicable in that model.

(*End of definition for* \l__color_current_tl.)

\color_group_begin:
\color_group_end:

Grouping for color is the same as using the basic \group_begin: and \group_end: functions. However, for semantic reasons, they are renamed here.

```
\cs_new_eq:NN \color_group_begin: \group_begin:
\cs_new_eq:NN \color_group_end:   \group_end:
```

(*End of definition for* \color_group_begin: *and* \color_group_end:. *These functions are documented on page* *320*.)

\color_ensure_current: A driver-independent wrapper for setting the foreground color to the current color "now".

```
\cs_new_protected:Npn \color_ensure_current:
  { \__color_select:N \l__color_current_tl }
```

(*End of definition for* `\color_ensure_current:`. *This function is documented on page 320.*)

\s__color_stop Internal scan marks.

```
\scan_new:N \s__color_stop
```

(*End of definition for* `\s__color_stop`.)

__color_select:N
__color_select_math:N
__color_select:nn

Take an internal color specification and pass it to the driver. This code is needed to ensure the current color but will also be used by the higher-level material.

```
\cs_new_protected:Npn \__color_select:N #1
  {
    \exp_after:wN \__color_select:nn #1
    \group_insert_after:N \__color_backend_reset:
  }
\cs_new_protected:Npn \__color_select_math:N #1
  { \exp_after:wN \__color_select:nn #1 }
\cs_new_protected:Npn \__color_select:nn #1#2
  { \use:c { __color_backend_select_ #1 :n } {#2} }
```

(*End of definition for* `\__color_select:N`, `\__color_select_math:N`, *and* `\__color_select:nn`.)

\l__color_current_tl The current color, with the model and

```
\tl_new:N \l__color_current_tl
\tl_set:Nn \l__color_current_tl { { gray } { 0 } }
```

(*End of definition for* `\l__color_current_tl`.)

93.2 Predefined color names

The ability to predefine colors with a name is a key part of this module and means there has to be a method for storing the results. At first sight, it seems natural to follow the usual expl3 model and create a `color` variable type for the process. That would then allow both local and global colors, constant colors and the like. However, these names need to be accessible in some form at the user level, for selection of colors either simply by name or as part of a more complex expression. This does not require that the full name is exposed but does require that they can be looked up in a predictable way. As such, it is more useful to expose just the color names as part of the interface, with the result that only local color names can be created. (This is also seen for example in key creation in l3keys.) As a result, color names are declarative (no `new` functions).

Since there is no need to manipulate colors *en masse*, each is stored in a two-part structure: a `prop` for the colors themselves, and a `tl` for the default model for each color.

93.3 Setup

\l__color_internal_int
\l__color_internal_tl

```
\int_new:N \l__color_internal_int
\tl_new:N \l__color_internal_tl
```

(*End of definition for* `\l__color_internal_int` *and* `\l__color_internal_tl`*.*)

\s__color_mark

Internal scan marks. `\s__color_stop` is already defined in l3color-base.

```
\scan_new:N \s__color_mark
```

(*End of definition for* `\s__color_mark`*.*)

\l__color_ignore_error_bool

Used to avoid issuing multiple errors if there is a change-of-model with input container an error.

```
\bool_new:N \l__color_ignore_error_bool
```

(*End of definition for* `\l__color_ignore_error_bool`*.*)

93.4 Utility functions

\color_if_exist_p:n
\color_if_exist:nTF

A simple wrapper to avoid needing to have the lookup repeated in too many places.To guard against a color created in a group, we need to test for entries in the `prop`.

```
\prg_new_conditional:Npnn \color_if_exist:n #1 { p , T, F, TF }
  {
    \prop_if_exist:cTF { l__color_named_ #1 _prop }
      {
        \prop_if_empty:cTF { l__color_named_ #1 _prop }
          \prg_return_false:
          \prg_return_true:
      }
      \prg_return_false:
  }
```

(*End of definition for* `\color_if_exist:nTF`*. This function is documented on page 323.*)

__color_model:N
__color_values:N

Simple abstractions.

```
\cs_new:Npn \__color_model:N #1 { \exp_after:wN \use_i:nn #1 }
\cs_new:Npn \__color_values:N #1 { \exp_after:wN \use_ii:nn #1 }
```

(*End of definition for* `\__color_model:N` *and* `\__color_values:N`*.*)

__color_extract:nNN
__color_extract:VNN

Recover the values for the standard model for a color.

```
\cs_new_protected:Npn \__color_extract:nNN #1#2#3
  {
    \tl_set_eq:Nc #2 { l__color_named_ #1 _tl }
    \prop_get:cVN { l__color_named_ #1 _prop } #2 #3
  }
\cs_generate_variant:Nn \__color_extract:nNN { V }
```

(*End of definition for* `\__color_extract:nNN`*.*)

93.5 Model conversion

__color_convert:nnN
__color_convert:VVN
__color_convert:nnnN
__color_convert:nVnN
__color_convert:nnVN
__color_convert_gray_gray:w
__color_convert_gray_rgb:w
__color_convert_gray_cmyk:w
__color_convert_cmyk_gray:w
__color_convert_cmyk_rgb:w
__color_convert_cmyk_cmyk:w
__color_convert_rgb_gray:w
__color_convert_rgb_rgb:w
__color_convert_rgb_cmyk:w
__color_convert_rgb_cmyk:nnn
__color_convert_rgb_cmyk:nnnn

Model conversion is carried out using standard formulae for base models, as described in the manual for xcolor (see also the *PostScript Language Reference Manual*). For other models direct conversion might not be defined, so we go through the fallback models if necessary.

```
\cs_new_protected:Npn \__color_convert:nnN #1#2#3
  { \__color_convert:nnVN {#1} {#2} #3 #3 }
\cs_generate_variant:Nn \__color_convert:nnN { VV }
\cs_generate_variant:Nn \exp_last_unbraced:Nf { c }
\cs_new_protected:Npn \__color_convert:nnnN #1#2#3#4
  {
    \tl_set:Ne #4
      {
        \cs_if_exist_use:cTF { __color_convert_ #1 _ #2 :w }
          { #3 \s__color_stop }
          {
            \cs_if_exist:cTF { __color_convert_ \use:c { c__color_fallback_ #1 _tl } _ #2 :
              {
                \exp_last_unbraced:cf
                  { __color_convert_ \use:c { c__color_fallback_ #1 _tl } _ #2 :w }
                  { \use:c { __color_convert_ #1 _ \use:c { c__color_fallback_ #1 _tl } :w
                  \s__color_stop
              }
              {
                \exp_last_unbraced:cf
                  { __color_convert_ \use:c { c__color_fallback_ #2 _tl } _ #2 :w }
                  {
                    \cs_if_exist_use:cTF { __color_convert_ #1 _ \use:c { c__color_fallback
                      { #3 \s__color_stop }
                      {
                        \exp_last_unbraced:cf
                          { __color_convert_ \use:c { c__color_fallback_ #1 _tl } _ \use:c
                          { \use:c { __color_convert_ #1 _ \use:c { c__color_fallback_ #1 _
                          \s__color_stop
                      }
                  }
                  \s__color_stop
              }
          }
      }
  }
\cs_generate_variant:Nn \__color_convert:nnnN { nV , nnV }
\cs_new:Npn \__color_convert_gray_gray:w #1 \s__color_stop
  { #1 }
\cs_new:Npn \__color_convert_gray_rgb:w #1 \s__color_stop
  { #1 ~ #1 ~ #1 }
\cs_new:Npn \__color_convert_gray_cmyk:w #1 \s__color_stop
  { 0 ~ 0 ~ 0 ~ \fp_eval:n { 1 - #1 } }
```

These rather odd values are based on NTSC television: the set are used for the cmyk conversion.

```
\cs_new:Npn \__color_convert_rgb_gray:w #1 ~ #2 ~ #3 \s__color_stop
  { \fp_eval:n { 0.3 * #1 + 0.59 * #2 + 0.11 * #3 } }
```

```
\cs_new:Npn \__color_convert_rgb_rgb:w #1 \s__color_stop
  { #1 }
```

The conversion from rgb to cmyk is the most complex: a two-step procedure which requires *black generation* and *undercolor removal* functions. The PostScript reference describes them as device-dependent, but following xcolor we assume they are linear. Moreover, as the likelihood of anyone using a non-unitary matrix here is tiny, we simplify and treat those two concepts as no-ops. To allow code sharing with parsing of cmy values, we have an intermediate function here (__color_convert_rgb_cmyk:nnn) which actually takes cmy values as input.

```
\cs_new:Npn \__color_convert_rgb_cmyk:w #1 ~ #2 ~ #3 \s__color_stop
  {
    \exp_args:Neee \__color_convert_rgb_cmyk:nnn
      { \fp_eval:n { 1 - #1 } }
      { \fp_eval:n { 1 - #2 } }
      { \fp_eval:n { 1 - #3 } }
  }
\cs_new:Npn \__color_convert_rgb_cmyk:nnn #1#2#3
  {
    \exp_args:Ne \__color_convert_rgb_cmyk:nnnn
      { \fp_eval:n { min( #1, #2 , #3 ) } } {#1} {#2} {#3}
  }
\cs_new:Npn \__color_convert_rgb_cmyk:nnnn #1#2#3#4
  {
    \fp_eval:n { min ( 1 , max ( 0 , #2 - #1 ) ) } \c_space_tl
    \fp_eval:n { min ( 1 , max ( 0 , #3 - #1 ) ) } \c_space_tl
    \fp_eval:n { min ( 1 , max ( 0 , #4 - #1 ) ) } \c_space_tl
    #1
  }
\cs_new:Npn \__color_convert_cmyk_gray:w #1 ~ #2 ~ #3 ~ #4 \s__color_stop
  { \fp_eval:n { 1 - min ( 1 , 0.3 * #1 + 0.59 * #2 + 0.11 * #3 + #4 ) } }
\cs_new:Npn \__color_convert_cmyk_rgb:w #1 ~ #2 ~ #3 ~ #4 \s__color_stop
  {
    \fp_eval:n { 1 - min ( 1 , #1 + #4 ) } \c_space_tl
    \fp_eval:n { 1 - min ( 1 , #2 + #4 ) } \c_space_tl
    \fp_eval:n { 1 - min ( 1 , #3 + #4 ) }
  }
\cs_new:Npn \__color_convert_cmyk_cmyk:w #1 \s__color_stop
  { #1 }
```

(*End of definition for* __color_convert:nnN *and others.*)

93.6 Color expressions

\l__color_model_tl
\l__color_value_tl
\l__color_next_model_tl
\l__color_next_value_tl

Working space to store the color data whilst doing calculations: keeping it on the stack is attractive but gets tricky (return is non-trivial).

```
\tl_new:N \l__color_model_tl
\tl_new:N \l__color_value_tl
\tl_new:N \l__color_next_model_tl
\tl_new:N \l__color_next_value_tl
```

(*End of definition for* \l__color_model_tl *and others.*)

__color_parse:nN
__color_parse_aux:nN
__color_parse_eq:Nn
__color_parse_eq:nNn
__color_parse:Nw
__color_parse_loop_init:Nnn
__color_parse_loop:w
__color_parse_loop_check:nn
__color_parse_loop:nn
__color_parse_gray:n
__color_parse_std:n
__color_parse_break:w
__color_parse_end:
__color_parse_mix:Nnnn
__color_parse_mix:NVVn
__color_parse_mix:nNnn
__color_parse_mix_gray:nw
__color_parse_mix_rgb:nw
__color_parse_mix_cmyk:nw

The main function for parsing color expressions removes actives but otherwise expands, then starts working through the expression itself. At the end, we apply the payload.

```
\cs_new_protected:Npe \__color_parse:nN #1#2
  {
    \tl_set:Ne \exp_not:c { l__color_named_ . _tl }
      { \exp_not:N \__color_model:N \exp_not:N \l__color_current_tl }
    \prop_put:NVe \exp_not:c { l__color_named_ . _prop }
      \exp_not:c { l__color_named_ . _tl }
      { \exp_not:N \__color_values:N \exp_not:N \l__color_current_tl }
    \exp_not:N \exp_args:Ne \exp_not:N \__color_parse_aux:nN
      { \exp_not:N \tl_to_str:n {#1} } #2
  }
```

Before going to all of the effort of parsing an expression, these two precursor functions look for a pre-defined name, either on its own or with a trailing ! (which is the same thing).

```
\cs_new_protected:Npn \__color_parse_aux:nN #1#2
  {
    \color_if_exist:nTF {#1}
      { \__color_parse_set_eq:Nn #2 {#1} }
      { \__color_parse:Nw #2#1 ! \s__color_stop }
    \__color_check_model:N #2
  }
\cs_new_protected:Npn \__color_parse_set_eq:Nn #1#2
  {
    \tl_if_empty:NTF \l_color_fixed_model_tl
      { \exp_args:Nv \__color_parse_set_eq:nNn { l__color_named_ #2 _tl } }
      { \exp_args:NV \__color_parse_set_eq:nNn \l_color_fixed_model_tl }
        #1 {#2}
  }
```

Here, we have to allow for the case where there is a fixed model: that can't be swept up by generic conversion as we are dealing with a named color.

```
\cs_new_protected:Npn \__color_parse_set_eq:nNn #1#2#3
  {
    \prop_get:cnNTF
      { l__color_named_ #3 _prop } {#1}
      \l__color_value_tl
      { \tl_set:Ne #2 { {#1} { \l__color_value_tl } } }
      {
        \tl_set_eq:Nc \l__color_model_tl { l__color_named_ #3 _tl }
        \prop_get:cVN { l__color_named_ #3 _prop } \l__color_model_tl
          \l__color_value_tl
        \__color_convert:nnN
          \l__color_model_tl {#1} \l__color_value_tl
        \tl_set:Ne #2
          {
            {#1}
            { \l__color_value_tl }
          }
      }
  }
\cs_new_protected:Npn \__color_parse:Nw #1#2 ! #3 \s__color_stop
  {
```

```
    \color_if_exist:nTF {#2}
      {
        \tl_if_blank:nTF {#3}
          { \__color_parse_set_eq:Nn #1 {#2} }
          { \__color_parse_loop_init:Nnn #1 {#2} {#3} }
      }
      {
        \msg_error:nnn { color } { unknown-color } {#2}
        \tl_set:Nn \l__color_current_tl { { gray } { 0 } }
      }
  }
```

Once we establish that a full parse is needed, the next job is to get the detail of the first color. That will determine the model we use for the calculation: splitting here makes checking that a bit easier.

```
\cs_new_protected:Npn \__color_parse_loop_init:Nnn #1#2#3
  {
    \group_begin:
      \__color_extract:nNN {#2} \l__color_model_tl \l__color_value_tl
      \__color_parse_loop:w #3 ! ! ! ! \s__color_stop
      \tl_set:Ne \l__color_internal_tl
        { { \l__color_model_tl } { \l__color_value_tl } }
    \exp_args:NNNV \group_end:
    \tl_set:Nn #1 \l__color_internal_tl
  }
```

This is the loop proper: there can be an open-ended set of colors to parse, separated by `!` tokens. There are a few cases to look out for. At the end of the expression and with we find a mix of 100 then we simply skip the next color entirely (we can't stop the loop as there might be a further valid color to mix in). On the other hand, if we get a mix of 0 then drop everything so far and start again. There is also a trailing `white` to "read in" if the final explicit data is a mix. Those conditions are separate from actually looping, which is therefore sorted out by checking if we have further data to process: in contrast to xcolor, we don't allow `!!` so the test can be simplified.

```
\cs_new_protected:Npn \__color_parse_loop:w #1 ! #2 ! #3 ! #4 ! #5 \s__color_stop
  {
    \tl_if_blank:nF {#1}
      {
        \bool_lazy_and:nnTF
          { \fp_compare_p:nNn {#1} > { 0 } }
          { \fp_compare_p:nNn {#1} < { 100 } }
          {
            \use:e
              {
                \__color_parse_loop:nn {#1}
                  { \tl_if_blank:nTF {#2} { white } {#2} }
              }
          }
          { \__color_parse_loop_check:nn {#1} {#2} }
      }
    \tl_if_blank:nF {#3}
      { \__color_parse_loop:w #3 ! #4 ! #5 \s__color_stop }
    \__color_parse_end:
  }
```

As these are unusual cases, we accept slower performance here for clearer code: check for the error conditions, handle the boundary cases after that.

```
\cs_new_protected:Npn \__color_parse_loop_check:nn #1#2
  {
    \bool_if:NF \l__color_ignore_error_bool
      {
        \bool_lazy_or:nnT
          { \fp_compare_p:nNn {#1} < { 0 } }
          { \fp_compare_p:nNn {#1} > { 100 } }
          { \msg_error:nnnnn { color } { out-of-range } {#1} { 0 } { 100 } }
      }
    \fp_compare:nNnF {#1} > \c_zero_fp
      {
        \tl_if_blank:nTF {#2}
          { \__color_extract:nNN { white } }
          { \__color_extract:nNN {#2} }
            \l__color_model_tl \l__color_value_tl
      }
  }
```

The "payload" of calculation in the loop first. If the model for the upcoming color is different from that of the existing (partial) color, convert the model. For `gray` the two are flipped round so that the outcome is something with "real" color. We are then in a position to do the actual calculation itself. The two auxiliaries here give us a way to break the loop should an invalid name be found.

```
\cs_new_protected:Npn \__color_parse_loop:nn #1#2
  {
    \color_if_exist:nTF {#2}
      {
        \__color_extract:nNN {#2} \l__color_next_model_tl \l__color_next_value_tl
        \tl_if_eq:NNF \l__color_model_tl \l__color_next_model_tl
          {
            \str_if_eq:VnTF \l__color_model_tl { gray }
              { \__color_parse_gray:n {#2} }
              { \__color_parse_std:n {#2} }
          }
        \tl_set:Ne \l__color_value_tl
          {
            \__color_parse_mix:NVVn
              \l__color_model_tl \l__color_value_tl \l__color_next_value_tl {#1}
          }
      }
      {
        \msg_error:nnn { color } { unknown-color } {#2}
        \__color_extract:nNN { black } \l__color_model_tl \l__color_value_tl
        \__color_parse_break:w
      }
  }
```

The `gray` model needs special handling: the models need to be swapped: we do that using a dedicated function.

```
\cs_new_protected:Npn \__color_parse_gray:n #1
  {
    \tl_set_eq:NN \l__color_model_tl \l__color_next_model_tl
```

```
    \tl_set:Nn \l__color_next_model_tl { gray }
    \exp_args:NnV \__color_convert:nnN { gray } \l__color_model_tl
      \l__color_value_tl
    \prop_get:cVN { l__color_named_ #1 _prop } \l__color_model_tl
      \l__color_next_value_tl
  }
\cs_new_protected:Npn \__color_parse_std:n #1
  {
    \prop_get:cVNF { l__color_named_ #1 _prop }
      \l__color_model_tl
      \l__color_next_value_tl
        {
          \__color_convert:VVN
            \l__color_next_model_tl
            \l__color_model_tl
            \l__color_next_value_tl
        }
  }
\cs_new_protected:Npn \__color_parse_break:w #1 \__color_parse_end: { }
\cs_new_protected:Npn \__color_parse_end: { }
```

Do the vector arithmetic: mainly a question of shuffling input, along with one pre-calculation to keep down the use of division.

```
\cs_new:Npn \__color_parse_mix:Nnnn #1#2#3#4
  {
    \exp_args:Nf \__color_parse_mix:nNnn
      { \fp_eval:n { #4 / 100 } }
      #1 {#2} {#3}
  }
\cs_generate_variant:Nn \__color_parse_mix:Nnnn { NVV }
\cs_new:Npn \__color_parse_mix:nNnn #1#2#3#4
  {
    \use:c { __color_parse_mix_ #2 :nw } {#1}
      #3 \s__color_mark #4 \s__color_stop
  }
\cs_new:Npn \__color_parse_mix_gray:nw #1#2 \s__color_mark #3 \s__color_stop
  { \fp_eval:n { #2 * #1 + #3 * ( 1 - #1 ) } }
\cs_new:Npn \__color_parse_mix_rgb:nw
  #1#2 ~ #3 ~ #4 \s__color_mark #5 ~ #6 ~ #7 \s__color_stop
  {
    \fp_eval:n { #2 * #1 + #5 * ( 1 - #1 ) } \c_space_tl
    \fp_eval:n { #3 * #1 + #6 * ( 1 - #1 ) } \c_space_tl
    \fp_eval:n { #4 * #1 + #7 * ( 1 - #1 ) }
  }
\cs_new:Npn \__color_parse_mix_cmyk:nw
  #1#2 ~ #3 ~ #4 ~ #5 \s__color_mark #6 ~ #7 ~ #8 ~ #9 \s__color_stop
  {
    \fp_eval:n { #2 * #1 + #6 * ( 1 - #1 ) } \c_space_tl
    \fp_eval:n { #3 * #1 + #7 * ( 1 - #1 ) } \c_space_tl
    \fp_eval:n { #4 * #1 + #8 * ( 1 - #1 ) } \c_space_tl
    \fp_eval:n { #5 * #1 + #9 * ( 1 - #1 ) }
  }
```

(*End of definition for* `\__color_parse:nN` *and others.*)

__color_parse_model_gray:w
__color_parse_model_rgb:w
__color_parse_model_cmyk:w
__color_parse_number:n
__color_parse_number:w

Turn the input into internal form, also tidying up the number quickly.

```
\cs_new:Npn \__color_parse_model_gray:w #1 , #2 \s__color_stop
  { { gray } { \__color_parse_number:n {#1} } }
\cs_new:Npn \__color_parse_model_rgb:w #1 , #2 , #3 , #4 \s__color_stop
  {
    { rgb }
    {
      \__color_parse_number:n {#1} ~
      \__color_parse_number:n {#2} ~
      \__color_parse_number:n {#3}
    }
  }
\cs_new:Npn \__color_parse_model_cmyk:w #1 , #2 , #3 , #4 , #5 \s__color_stop
  {
    { cmyk }
    {
      \__color_parse_number:n {#1} ~
      \__color_parse_number:n {#2} ~
      \__color_parse_number:n {#3} ~
      \__color_parse_number:n {#4}
    }
  }
\cs_new:Npn \__color_parse_number:n #1
  {  \__color_parse_number:w #1 . 0 . \s__color_stop }
\cs_new:Npn \__color_parse_number:w #1 . #2 . #3 \s__color_stop
  { \tl_if_blank:nTF {#1} { 0 } {#1} . #2 }
```

(*End of definition for* `\__color_parse_model_gray:w` *and others.*)

__color_parse_model_Gray:w
__color_parse_model_hsb:w
__color_parse_model_Hsb:w
__color_parse_model_HSB:w
__color_parse_model_HTML:w
__color_parse_model_RGB:w
__color_parse_model_hsb:nnn
__color_parse_model_hsb_aux:nnn
__color_parse_model_hsb:nnnn
__color_parse_model_hsb:nnnnn
__color_parse_model_hsb_0:nnnn
__color_parse_model_hsb_1:nnnn
__color_parse_model_hsb_2:nnnn
__color_parse_model_hsb_3:nnnn
__color_parse_model_hsb_4:nnnn
__color_parse_model_hsb_5:nnnn
__color_parse_model_wave:w
__color_parse_model_wave_auxi:nn
__color_parse_model_wave_auxii:nn
__color_parse_model_wave_rho:n

```
\cs_new:Npn \__color_parse_model_Gray:w #1 , #2 \s__color_stop
  { { gray } { \fp_eval:n { #1 / 15 } } }
\cs_new:Npn \__color_parse_model_hsb:w #1 , #2 , #3 , #4 \s__color_stop
  { \__color_parse_model_hsb:nnn {#1} {#2} {#3} }
\cs_new:Npn \__color_parse_model_Hsb:w #1 , #2 , #3 , #4 \s__color_stop
  {
    \exp_args:Ne \__color_parse_model_hsb:nnn { \fp_eval:n { #1 / 360 } }
      {#2} {#3}
  }
```

The conversion here is non-trivial but is described at length in the xcolor manual. For ease, we calculate the integer and fractional parts of the hue first, then use them to work out the possible values for r, g and b before putting them in the correct places.

```
\cs_new:Npn \__color_parse_model_hsb:nnn #1#2#3
  {
    { rgb }
    {
      \exp_args:Ne \__color_parse_model_hsb_aux:nnn
        { \fp_eval:n { 6 * (#1) } } {#2} {#3}
    }
  }
\cs_new:Npn \__color_parse_model_hsb_aux:nnn #1#2#3
  {
    \exp_args:Nee \__color_parse_model_hsb_aux:nnnn
```

```
      { \fp_eval:n { floor(#1) } } { \fp_eval:n { #1 - floor(#1) } }
      {#2} {#3}
  }
\cs_new:Npn \__color_parse_model_hsb_aux:nnnn #1#2#3#4
  {
    \use:e
      {
        \exp_not:N \__color_parse_model_hsb_aux:nnnnn
          { \__color_parse_number:n {#4} }
          { \fp_eval:n { round(#4 * (1 - #3) ,5) } }
          { \fp_eval:n { round(#4 * ( 1 - #3 * #2 ) ,5) } }
          { \fp_eval:n { round(#4 * ( 1 - #3 * (1 - #2) ) ,5) } }
          {#1}
      }
  }
\cs_new:Npn \__color_parse_model_hsb_aux:nnnnn #1#2#3#4#5
  { \use:c { __color_parse_model_hsb_ #5 :nnnn } {#1} {#2} {#3} {#4} }
\cs_new:cpn { __color_parse_model_hsb_0:nnnn } #1#2#3#4 { #1 ~ #4 ~ #2 }
\cs_new:cpn { __color_parse_model_hsb_1:nnnn } #1#2#3#4 { #3 ~ #1 ~ #2 }
\cs_new:cpn { __color_parse_model_hsb_2:nnnn } #1#2#3#4 { #2 ~ #1 ~ #4 }
\cs_new:cpn { __color_parse_model_hsb_3:nnnn } #1#2#3#4 { #2 ~ #3 ~ #1 }
\cs_new:cpn { __color_parse_model_hsb_4:nnnn } #1#2#3#4 { #4 ~ #2 ~ #1 }
\cs_new:cpn { __color_parse_model_hsb_5:nnnn } #1#2#3#4 { #1 ~ #2 ~ #3 }
\cs_new:cpn { __color_parse_model_hsb_6:nnnn } #1#2#3#4 { #1 ~ #2 ~ #2 }
\cs_new:Npn \__color_parse_model_HSB:w #1 , #2 , #3 , #4 \s__color_stop
  {
    \exp_args:Neee \__color_parse_model_hsb:nnn
      { \fp_eval:n { round((#1) / 240,5) } }
      { \fp_eval:n { round((#2) / 240,5) } }
      { \fp_eval:n { round((#3) / 240,5) } }
  }
\cs_new:Npn \__color_parse_model_HTML:w #1 , #2 \s__color_stop
  { \__color_parse_model_HTML_aux:w #1 0 0 0 0 0 0 \s__color_stop }
\cs_new:Npn \__color_parse_model_HTML_aux:w #1#2#3#4#5#6#7 \s__color_stop
  {
    { rgb }
    {
      \fp_eval:n { round(\int_from_hex:n {#1#2} / 255,5) } ~
      \fp_eval:n { round(\int_from_hex:n {#3#4} / 255,5) } ~
      \fp_eval:n { round(\int_from_hex:n {#5#6} / 255,5) }
    }
  }
\cs_new:Npn \__color_parse_model_RGB:w #1 , #2 , #3 , #4 \s__color_stop
  {
    { rgb }
    {
      \fp_eval:n { round((#1) / 255,5) } ~
      \fp_eval:n { round((#2) / 255,5) } ~
      \fp_eval:n { round((#3) / 255,5) }
    }
  }
```

Following the description in the xcolor manual. As we always use `rgb`, there is no need to find the sixth, we just pas the information straight to the `hsb` auxiliary defined earlier.

```
\cs_new:Npn \__color_parse_model_wave:w #1 , #2 \s__color_stop
  {
    { rgb }
    {
      \fp_compare:nNnTF {#1} < { 420 }
        { \__color_parse_model_wave_auxi:nn {#1} { 0.3 + 0.7 * (#1 - 380) / 40 }
        }
        {
          \fp_compare:nNnTF {#1} > { 700 }
            { \__color_parse_model_wave_auxi:nn {#1} { 0.3 + 0.7 * (#1 - 780) / -80 } }
            { \__color_parse_model_wave_auxi:nn {#1} { 1 } }
        }
    }
  }
\cs_new:Npn \__color_parse_model_wave_auxi:nn #1#2
  {
    \fp_compare:nNnTF {#1} < { 440 }
      {
        \__color_parse_model_wave_auxii:nn
          { 4 + \__color_parse_model_wave_rho:n { (#1 - 440) / -60 } }
          {#2}
      }
      {
        \fp_compare:nNnTF {#1} < { 490 }
          {
            \__color_parse_model_wave_auxii:nn
              { 4 - \__color_parse_model_wave_rho:n { (#1 - 440) / 50 } }
              {#2}
          }
          {
            \fp_compare:nNnTF {#1} < { 510 }
              {
                \__color_parse_model_wave_auxii:nn
                  { 2 + \__color_parse_model_wave_rho:n { (#1 - 510) / -20 } }
                  {#2}
              }
              {
                \fp_compare:nNnTF {#1} < { 580 }
                  {
                    \__color_parse_model_wave_auxii:nn
                      { 2 - \__color_parse_model_wave_rho:n { (#1 - 510) / 70 } }
                      {#2}
                  }
                  {
                    \fp_compare:nNnTF {#1} < { 645 }
                      {
                        \__color_parse_model_wave_auxii:nn
                          { \__color_parse_model_wave_rho:n { (#1 - 645) / -65 } }
                          {#2}
                      }
                      { \__color_parse_model_wave_auxii:nn { 0 } {#2} }
                  }
              }
          }
```

```
      }
  }
\cs_new:Npn \__color_parse_model_wave_auxii:nn #1#2
  {
    \exp_args:Neee \__color_parse_model_hsb_aux:nnn
      { \fp_eval:n {#1} }
      { 1 }
      { \__color_parse_model_wave_rho:n {#2} }
  }
\cs_new:Npn \__color_parse_model_wave_rho:n #1
  { \fp_eval:n { min(1, max(0,#1) ) } }
```

(*End of definition for* `\__color_parse_model_Gray:w` *and others.*)

`\__color_parse_model_cmy:w`

Simply pass data to the conversion functions.

```
\cs_new:Npn \__color_parse_model_cmy:w #1 , #2 , #3 , #4 \s__color_stop
  {
    { cmyk }
    { \__color_convert_rgb_cmyk:nnn {#1} {#2} {#3} }
  }
```

(*End of definition for* `\__color_parse_model_cmy:w`*.*)

`\__color_parse_model_tHsb:w`
`\__color_parse_model_tHsb:n`
`\__color_parse_model_tHsb:nw`

There are three stages to the process here: bring the `tH` argument into the normal range, divide through to get to `hsb` and finally convert that to `rgb`. The final stage can be delegated to the parsing function for `hsb`, and the conversion from `Hsb` to `hsb` is trivial, so the main focus here is the first stage. We use a simple expandable loop to do the work, and we implement the equation given in the xcolor manual (number 85 there) as a simple expression.

```
\cs_new:Npn \__color_parse_model_tHsb:w #1 , #2 , #3 , #4 \s__color_stop
  {
    \exp_args:Ne \__color_parse_model_hsb:nnn
      { \__color_parse_model_tHsb:n {#1} } {#2} {#3}
  }
\cs_new:Npn \__color_parse_model_tHsb:n #1
  {
    \__color_parse_model_tHsb:nw {#1}
        0 ,   0 ;
       60 ,  30 ;
      120 ,  60 ;
      180 , 120 ;
      210 , 180 ;
      240 , 240 ;
      360 , 360 ;
      \q_recursion_tail , ;
      \q_recursion_stop
  }
\cs_new:Npn \__color_parse_model_tHsb:nw #1 #2 , #3 ; #4 , #5 ;
  {
    \quark_if_recursion_tail_stop_do:nn {#4} { 0 }
    \fp_compare:nNnTF {#1} > {#4}
      { \__color_parse_model_tHsb:nw {#1} #4 , #5 ; }
      {
        \use_i_delimit_by_q_recursion_stop:nw
```

```
        { \fp_eval:n { ((#1 - #2) / (#4 - #2) * (#5 - #3) + #3) / 360 } }
    }
  }
```

(*End of definition for* `\__color_parse_model_tHsb:w`, `\__color_parse_model_tHsb:n`, *and* `\__color_parse_model_tHsb:nw`.)

`\__color_parse_model_&spot:w` We cannot extract data here from that passed by xcolor, so we fall back on a black tint.

```
\cs_new:cpn { __color_parse_model_&spot:w } #1 , #2 \s__color_stop
  { { gray } { #1 } }
```

(*End of definition for* `\__color_parse_model_&spot:w`.)

93.7 Selecting colors (and color models)

`\l_color_fixed_model_tl` For selecting a single fixed model.

```
\tl_new:N \l_color_fixed_model_tl
```

(*End of definition for* `\l_color_fixed_model_tl`. *This variable is documented on page 323.*)

`\__color_check_model:N` `\__color_check_model:nn` Check that the model in use is the one required.

```
\cs_new_protected:Npn \__color_check_model:N #1
  {
    \tl_if_empty:NF \l_color_fixed_model_tl
      {
        \exp_after:wN \__color_check_model:nn #1
        \tl_if_eq:NNF \l__color_model_tl \l_color_fixed_model_tl
          {
            \__color_convert:VVN \l__color_model_tl \l_color_fixed_model_tl
              \l__color_value_tl
          }
        \tl_set:Ne #1
          { { \l_color_fixed_model_tl } { \l__color_value_tl } }
      }
  }
\cs_new_protected:Npn \__color_check_model:nn #1#2
  {
    \tl_set:Nn \l__color_model_tl {#1}
    \tl_set:Nn \l__color_value_tl {#2}
  }
```

(*End of definition for* `\__color_check_model:N` *and* `\__color_check_model:nn`.)

`\__color_finalise_current:` A backend-neutral location for "last minute" manipulations before handing off to the backend code. We set the special . syntax here: this will therefore always be available. The finalisation is separate from the main function so it can also be applied to *e.g.* page color.

```
\cs_new_protected:Npe \__color_finalise_current:
  {
    \tl_set:Ne \exp_not:c { l__color_named_ . _tl }
      { \exp_not:N \__color_model:N \exp_not:N \l__color_current_tl }
    \prop_clear:N \exp_not:c { l__color_named_ . _prop }
    \prop_put:NVe \exp_not:c { l__color_named_ . _prop }
      \exp_not:c { l__color_named_ . _tl }
```

```
      { \exp_not:N \__color_values:N \exp_not:N \l__color_current_tl }
  }
```

(*End of definition for* `\__color_finalise_current:`.)

`\color_select:n`
`\color_select:nn`
`\__color_select_main:Nw`
`\__color_select_loop:Nw`
`\__color_select:nnN`
`\__color_select_swap:Nnn`

Parse the input expressions then get the backend to actually activate them. The main complexity here is the need to check through multiple models. That is done “locally” here as the approach is subtly different to when different models are being stored.

```
\cs_new_protected:Npn \color_select:n #1
  {
    \__color_parse:nN {#1} \l__color_current_tl
    \__color_finalise_current:
    \__color_select:N \l__color_current_tl
  }
\cs_new_protected:Npn \color_select:nn #1#2
  {
    \__color_select_main:Nw \l__color_current_tl
      #1 / / \s__color_mark #2 / / \s__color_stop
    \__color_finalise_current:
    \__color_select:N \l__color_current_tl
  }
```

If the first color model is the fixed one, or if there is no fixed model, we don’t need most of the data: just set up and apply the backend function.

```
\cs_new_protected:Npn \__color_select_main:Nw
  #1 #2 / #3 / #4 \s__color_mark #5 / #6 / #7 \s__color_stop
  {
    \__color_select:nnN {#2} {#5} #1
    \bool_lazy_or:nnF
      { \tl_if_empty_p:N \l_color_fixed_model_tl }
      { \str_if_eq_p:nV {#2} \l_color_fixed_model_tl }
      { \__color_select_loop:Nw #1 #3 / #4 \s__color_mark #6 / #7 \s__color_stop }
  }
```

If a fixed model applies, we need to check each possible value in order. If there is no hit at all, fall back on the generic formula-based interchange.

```
\cs_new_protected:Npn \__color_select_loop:Nw
  #1 #2 / #3 \s__color_mark #4 / #5 \s__color_stop
  {
    \str_if_eq:nVTF {#2} \l_color_fixed_model_tl
      { \__color_select:nnN {#2} {#4} #1 }
      {
        \tl_if_blank:nTF {#2}
          { \exp_after:wN \__color_select_swap:Nnn \exp_after:wN #1 #1 }
          { \__color_select_loop:Nw #1 #3 \s__color_mark #5 \s__color_stop }
      }
  }
\cs_new_protected:Npn \__color_select:nnN #1#2#3
  {
    \cs_if_exist:cTF { __color_parse_model_ #1 :w }
      {
        \tl_set:Ne #3
          { \use:c { __color_parse_model_ #1 :w } #2 , 0 , 0 , 0 , 0 \s__color_stop }
      }
      { \msg_error:nnn { color } { unknown-model } {#1} }
```

```
  }
\cs_new_protected:Npn \__color_select_swap:Nnn #1#2#3
  {
    \__color_convert:nVnN {#2} \l_color_fixed_model_tl {#3} \l__color_value_tl
    \tl_set:Ne #1
      { { \l_color_fixed_model_tl } { \l__color_value_tl } }
  }
```

(*End of definition for* `\color_select:n` *and others. These functions are documented on page 323.*)

93.8 Math color

The approach here is the same as for the LaTeX 2_ε `\mathcolor` command, but as we are working at the expl3 level we can make some minor changes.

`\l_color_math_active_tl` Tokens representing active sub/superscripts.

```
\tl_new:N \l_color_math_active_tl
\tl_set:Nn \l_color_math_active_tl { ' }
```

(*End of definition for* `\l_color_math_active_tl`*. This function is documented on page 324.*)

`\g__color_math_seq` Not all engines have multiple color stacks, and at the same time we are not expecting breaking within a colored math fragment. So we track the color stack ourselves.

```
\seq_new:N \g__color_math_seq
```

(*End of definition for* `\g__color_math_seq`*.*)

`\color_math:nn`
`\color_math:nnn`
`\__color_math:nn`

The basic set up here is relatively simple: store the current color, parse the new color as-normal, then switch color before inserting the tokens we are asked to change. The tricky part is right at the end, handling the reset.

```
\cs_new_protected:Npn \color_math:nn #1#2
  {
    \__color_math:nn {#2}
      { \__color_parse:nN {#1} \l__color_current_tl }
  }
\cs_new_protected:Npn \color_math:nnn #1#2#3
  {
    \__color_math:nn {#3}
      {
        \__color_select_main:Nw \l__color_current_tl
          #1 / / \s__color_mark #2 / / \s__color_stop
      }
  }
\cs_new_protected:Npn \__color_math:nn #1#2
  {
    \seq_gpush:NV \g__color_math_seq \l__color_current_tl
    #2
    \__color_select_math:N \l__color_current_tl
    #1
    \__color_math_scan:w
  }
```

(*End of definition for* `\color_math:nn`*,* `\color_math:nnn`*, and* `\__color_math:nn`*. These functions are documented on page 324.*)

__color_math_scan:w
__color_math_scan_auxi:
__color_math_scan_auxii:
__color_math_scan_end:

The complication when changing the color back is due to the fact that the `\color_-math:nn(n)` may be followed by `^` or `_` or the hidden superscript (for example `'`) and its argument may end in a `\mathop` in which case the sub- and superscripts may be attached as `\limits` instead of after the material. All cases need separate treatment. To avoid repeatedly collecting the same token, we first check for an alignment tab: assuming we don't have one of those, we can "recycle" `\l_peek_token` safely. As we have an explicit `\c_alignment_token`, there needs to be an align-safe group present.

```
\cs_new_protected:Npn \__color_math_scan:w
  {
    \peek_remove_filler:n
      {
        \group_align_safe_begin:
        \peek_catcode:NTF \c_alignment_token
          {
            \group_align_safe_end:
            \__color_math_scan_end:
          }
          {
            \group_align_safe_end:
            \__color_math_scan_auxi:
          }
      }
  }
```

Dealing with literal `_` and `^` is easy, and as we have exactly two cases, we can hard-code this. We use a hard-coded list for limits: these are all primitives. The `\use_none:n` here also removes the test token so it is left just in the right place.

```
\cs_new_protected:Npn \__color_math_scan_auxi:
  {
    \token_case_catcode:NnTF \l_peek_token
      {
        \c_math_subscript_token   { }
        \c_math_superscript_token { }
      }
      { \__color_math_scripts:Nw }
      {
        \token_case_meaning:NnTF \l_peek_token
          {
            \tex_limits:D        { \tex_limits:D }
            \tex_nolimits:D      { \tex_nolimits:D }
            \tex_displaylimits:D { \tex_displaylimits:D }
          }
          { \__color_math_scan:w \use_none:n }
          { \__color_math_scan_auxii: }
      }
  }
```

The one final case to handle is math-active tokens, most obviously `'`, as these won't be covered earlier.

```
\cs_new_protected:Npn \__color_math_scan_auxii:
  {
    \tl_map_inline:Nn \l_color_math_active_tl
      {
        \token_if_eq_meaning:NNT \l_peek_token ##1
```

```
     {
      \tl_map_break:n
       {
        \use_i:nn
         { \__color_math_scan_auxiii:N ##1 }
       }
     }
    \__color_math_scan_end:
   }
 }
\cs_new_protected:Npn \__color_math_scan_auxiii:N #1
 {
  \exp_after:wN \exp_after:wN \exp_after:wN \__color_math_scan:w
   \char_generate:nn { `#1 } { 13 }
 }
\cs_new_protected:Npn \__color_math_scan_end:
 {
  \__color_backend_reset:
  \seq_gpop:NN \g__color_math_seq \l__color_current_tl
 }
```

(*End of definition for* `\__color_math_scan:w` *and others.*)

`\__color_math_scripts:Nw`
`\__color_math_script_aux:N`

The tricky part of handling sub and superscripts is that we have to reset color to the one that is on the stack but reset it back to what it was before to allow for cases like

```
\[  \color_math:n { red } { a + \sum } _ { i = 1 } ^ { n }  \]
```

Here, TeX constructs a `\vbox` stacking subscript, summation sign, and superscript. So technically the superscript comes first and the `\sum` that should get colored red is the middle.

The approach here is to set up a brace group immediately after the script token, then to set the color appropriately in that argument. We need an extra group to keep the color contained, and as we need to allow for an explicit closing brace in the source, the inner group also is a brace one rather than `\group_begin:`-based. At the end of the outer group we need to insert `\__color_math_scan:w` to continue the search for a second script token.

Notice that here we *don't* need to use the math-specific color selector as we can allow the `\group_insert_after:N \@@_backend_reset:` to operate normally.

```
\cs_new_protected:Npn \__color_math_scripts:Nw #1
 {
  #1
  \c_group_begin_token
   \c_group_begin_token
    \seq_get:NN \g__color_math_seq \l__color_current_tl
    \__color_select:N \l__color_current_tl
    \group_insert_after:N \c_group_end_token
    \group_insert_after:N \__color_math_scan:w
  \peek_remove_filler:n
   {
    \peek_catcode_remove:NF \c_group_begin_token
     { \__color_math_script_aux:N }
   }
 }
```

Deal with the case where we do not have an explicit brace pair in the source.

```
\cs_new_protected:Npn \__color_math_script_aux:N #1 { #1 \c_group_end_token }
```

(*End of definition for* `\__color_math_scripts:Nw` *and* `\__color_math_script_aux:N`.)

93.9 Fill and stroke color

`\color_fill:n` `\color_stroke:n` `\color_fill:nn` `\color_stroke:nn` `\__color_draw:nnn`

```
\cs_new_protected:Npn \color_fill:n #1
  {
    \__color_parse:nN {#1} \l__color_current_tl
    \exp_after:wN \__color_draw:nnn \l__color_current_tl { fill }
  }
\cs_new_protected:Npn \color_stroke:n #1
  {
    \__color_parse:nN {#1} \l__color_current_tl
    \exp_after:wN \__color_draw:nnn \l__color_current_tl { stroke }
  }
\cs_new_protected:Npn \color_fill:nn #1#2
  {
    \__color_select_main:Nw \l__color_current_tl
      #1 / / \s__color_mark #2 / / \s__color_stop
    \exp_after:wN \__color_draw:nnn \l__color_current_tl { fill }
  }
\cs_new_protected:Npn \color_stroke:nn #1#2
  {
    \__color_select_main:Nw \l__color_current_tl
      #1 / / \s__color_mark #2 / / \s__color_stop
    \exp_after:wN \__color_draw:nnn \l__color_current_tl { stroke }
  }
\cs_new_protected:Npn \__color_draw:nnn #1#2#3
  {
    \use:c { __color_backend_ #3 _ #1 :n } {#2}
    \exp_args:Nc \group_insert_after:N { __color_backend_ #3 _ reset: }
  }
```

(*End of definition for* `\color_fill:n` *and others. These functions are documented on page 324.*)

93.10 Defining named colors

`\l__color_named_tl` Space to store the detail of the named color.

```
\tl_new:N \l__color_named_tl
```

(*End of definition for* `\l__color_named_tl`.)

`\color_set:nn` `\__color_set:nnn` `\__color_set:nn` `\__color_set:nnw` `\color_set:nnn` `\__color_set_aux:nnn` `\__color_set_colon:nnw` `\__color_set_loop:nw` `\color_set_eq:nn`

Defining named colors means working through the model list and saving both the "main" color and any equivalents in other models. Even if there is only one model, we store a `prop` as well as a `tl`, as there could be grouping weirdness, etc. When setting using an expression, we need to avoid any fixed model issues, which is done without a group as in l3keys.

```
\cs_new_protected:Npn \color_set:nn #1#2
  {
```

```
    \exp_args:NV \__color_set:nnn
      \l_color_fixed_model_tl {#1} {#2}
  }
\cs_new_protected:Npn \__color_set:nnn #1#2#3
  {
    \tl_clear:N \l_color_fixed_model_tl
    \__color_set:nn {#2} {#3}
    \tl_set:Nn \l_color_fixed_model_tl {#1}
  }
\cs_new_protected:Npn \__color_set:nn #1#2
  {
    \str_if_eq:nnF {#1} { . }
      {
        \__color_parse:nN {#2} \l__color_named_tl
        \tl_clear_new:c { l__color_named_ #1 _tl }
        \tl_set:ce { l__color_named_ #1 _tl }
          { \__color_model:N \l__color_named_tl }
        \prop_clear_new:c { l__color_named_ #1 _prop }
        \prop_put:cve { l__color_named_ #1 _prop } { l__color_named_ #1 _tl }
          { \__color_values:N \l__color_named_tl }
        \__color_set:nnw {#1} {#2} #2 ! \s__color_stop
      }
  }
```

When setting an expression-based color, there could be multiple model data available for one or more of the input colors. Where that is true for the *first* named color in an expression, we re-parse the expression when they are also parameter-based: only `cmyk`, `gray` and `rgb` make any sense here. There is a bit of a performance hit but this should be rare and taking place during set-up.

```
\cs_new_protected:Npn \__color_set:nnw #1#2#3 ! #4 \s__color_stop
  {
    \clist_map_inline:nn { cmyk , gray , rgb }
      {
        \prop_get:cnNT { l__color_named_ #3 _prop } {##1} \l__color_internal_tl
          {
            \prop_if_in:cnF { l__color_named_ #1 _prop } {##1}
              {
                \group_begin:
                  \bool_set_true:N \l__color_ignore_error_bool
                  \tl_set:cn { l__color_named_ #3 _tl } {##1}
                  \__color_parse:nN {#2} \l__color_internal_tl
                \exp_args:NNNV \group_end:
                \tl_set:Nn \l__color_internal_tl \l__color_internal_tl
                \prop_put:cee { l__color_named_ #1 _prop }
                  { \__color_model:N \l__color_internal_tl }
                  { \__color_values:N \l__color_internal_tl }
              }
          }
      }
  }
\cs_new_protected:Npn \color_set:nnn #1#2#3
  {
    \str_if_eq:nnF {#1} { . }
      {
```

```
      \tl_clear_new:c { l__color_named_ #1 _tl }
      \prop_clear_new:c { l__color_named_ #1 _prop }
      \exp_args:Ne \__color_set_aux:nnn { \tl_to_str:n {#2} }
        {#1} {#3}
    }
  }
\cs_new_protected:Npe \__color_set_aux:nnn #1#2#3
  {
    \exp_not:N \__color_set_colon:nnw {#2} {#3}
      #1 \c_colon_str \c_colon_str \exp_not:N \s__color_stop
  }
\use:e
  {
    \cs_new_protected:Npn \exp_not:N \__color_set_colon:nnw
      #1#2 #3 \c_colon_str #4 \c_colon_str
      #5 \exp_not:N \s__color_stop
  }
  {
    \tl_if_blank:nTF {#4}
      { \__color_set_loop:nw {#1} #3 }
      { \__color_set_loop:nw {#1} #4 }
        / / \s__color_mark #2 / / \s__color_stop
  }
\cs_new_protected:Npn \__color_set_loop:nw
  #1#2 / #3 \s__color_mark #4 / #5 \s__color_stop
  {
    \tl_if_blank:nF {#2}
      {
        \__color_select:nnN {#2} {#4} \l__color_named_tl
        \tl_set:Ne \l__color_internal_tl { \__color_model:N \l__color_named_tl }
        \tl_if_empty:cT { l__color_named_ #1 _tl }
          { \tl_set_eq:cN { l__color_named_ #1 _tl } \l__color_internal_tl }
        \prop_put:cVe { l__color_named_ #1 _prop } \l__color_internal_tl
          { \__color_values:N \l__color_named_tl }
        \__color_set_loop:nw {#1} #3 \s__color_mark #5 \s__color_stop
      }
  }
\cs_new_protected:Npn \color_set_eq:nn #1#2
  {
    \color_if_exist:nTF {#2}
      {
        \tl_clear_new:c { l__color_named_ #1 _tl }
        \prop_clear_new:c { l__color_named_ #1 _prop }
        \str_if_eq:nnTF {#2} { . }
          {
            \tl_set:ce { l__color_named_ #1 _tl }
              { \__color_model:N \l__color_current_tl }
            \prop_put:cve { l__color_named_ #1 _prop } { l__color_named_ #1 _tl }
              { \__color_values:N \l__color_current_tl }
          }
          {
            \tl_set_eq:cc { l__color_named_ #1 _tl } { l__color_named_ #2 _tl }
            \prop_set_eq:cc { l__color_named_ #1 _prop } { l__color_named_ #2 _prop }
          }
```

```
      }
      {
        \msg_error:nnn { color } { unknown-color } {#2}
      }
  }
```

(*End of definition for* `\color_set:nn` *and others. These functions are documented on page 323.*)

A small set of colors are always defined.

```
\color_set:nnn { black } { gray } { 0 }
\color_set:nnn { white } { gray } { 1 }
\color_set:nnn { cyan }    { cmyk } { 1 , 0 , 0 , 0 }
\color_set:nnn { magenta } { cmyk } { 0 , 1 , 0 , 0 }
\color_set:nnn { yellow }  { cmyk } { 0 , 0 , 1 , 0 }
\color_set:nnn { red }   { rgb } { 1 , 0 , 0 }
\color_set:nnn { green } { rgb } { 0 , 1 , 0 }
\color_set:nnn { blue }  { rgb } { 0 , 0 , 1 }
```

`\l__color_named_._prop`
`\l__color_named_._tl`

A special named color: this is always defined though not fixed in definition.

```
\prop_new:c { l__color_named_._prop }
\tl_new:c { l__color_named_._tl }
\tl_set:ce { l__color_named_._tl } { \__color_model:N \l__color_current_tl }
```

(*End of definition for* `\l__color_named_._prop` *and* `\l__color_named_._tl`*.*)

93.11 Exporting colors

`\color_export:nnN`
`\color_export:nnnN`
`\__color_export:nN`
`\__color_export:nnnN`

```
\cs_new_protected:Npn \color_export:nnN #1#2#3
  {
    \group_begin:
      \tl_if_exist:cT { c__color_export_ #2 _tl }
        { \tl_set_eq:Nc \l_color_fixed_model_tl { c__color_export_ #2 _tl } }
      \__color_parse:nN {#1} #3
      \__color_export:nN {#2} #3
    \exp_args:NNNV \group_end:
    \tl_set:Nn #3 #3
  }
\cs_new_protected:Npn \color_export:nnnN #1#2#3#4
  {
    \__color_select_main:Nw #4
      #1 / / \s__color_mark #2 / / \s__color_stop
    \__color_export:nN {#3} #4
  }
\cs_new_protected:Npn \__color_export:nN #1#2
  { \exp_after:wN \__color_export:nnnN #2 {#1} #2 }
\cs_new:Npn \__color_export:nnnN #1#2#3#4
  {
    \cs_if_exist_use:cF { __color_export_format_ #3 :nnN }
      {
        \msg_error:nnn { color } { unknown-export-format } {#3}
        \use_none:nnn
      }
        {#1} {#2} #4
  }
```

(*End of definition for* `\color_export:nnN` *and others. These functions are documented on page* 325.)

__color_export_format_backend:nnN

Simple.

```
\cs_new_protected:Npn \__color_export_format_backend:nnN #1#2#3
  { \tl_set:Nn #3 { {#1} {#2} } }
```

(*End of definition for* `\__color_export_format_backend:nnN`.)

__color_export:nnnNN

A generic auxiliary for cases where only one model is appropriate.

```
\cs_new_protected:Npn \__color_export:nnnNN #1#2#3#4#5
  {
    \str_if_eq:nnTF {#2} {#1}
      { #5 #4 #3 \s__color_stop }
      {
        \__color_convert:nnnN {#2} {#1} {#3} #4
        \exp_after:wN #5 \exp_after:wN #4
          #4 \s__color_stop
      }
  }
```

(*End of definition for* `\__color_export:nnnNN`.)

\c__color_export_comma-sep-cmyk_tl
\c__color_export_comma-sep-rgb_tl
\c__color_export_HTML_tl
\c__color_export_space-sep-cmyk_tl
\c__color_export_space-sep-rgb_tl

```
\tl_const:cn { c__color_export_comma-sep-cmyk_tl } { cmyk }
\tl_const:cn { c__color_export_comma-sep-rgb_tl } { rgb }
\tl_const:Nn \c__color_export_HTML_tl { rgb }
\tl_const:cn { c__color_export_space-sep-cmyk_tl } { cmyk }
\tl_const:cn { c__color_export_space-sep-rgb_tl } { rgb }
```

(*End of definition for* `\c__color_export_comma-sep-cmyk_tl` *and others.*)

__color_export_format_comma-sep-cmyk:nnN
__color_export_format_comma-sep-rgb:nnN
__color_export_format_space-sep-cmyk:nnN
__color_export_format_space-sep-rgb:nnN

```
\group_begin:
  \cs_set_protected:Npn \__color_tmp:w #1#2
    {
      \cs_new_protected:cpe { __color_export_format_ #1 :nnN } ##1##2##3
        {
          \exp_not:N \__color_export:nnnNN {#2} {##1} {##2} ##3
            \exp_not:c { __color_export_ #1 :Nw }
        }
    }
  \__color_tmp:w { comma-sep-cmyk } { cmyk }
  \__color_tmp:w { comma-sep-rgb }  { rgb }
  \__color_tmp:w { HTML }           { rgb }
  \__color_tmp:w { space-sep-cmyk } { cmyk }
  \__color_tmp:w { space-sep-rgb }  { rgb }

\group_end:
```

(*End of definition for* `\__color_export_format_comma-sep-cmyk:nnN` *and others.*)

__color_export_space-sep-cmyk:Nw
__color_export_comma-sep-cmyk:Nw

```
\cs_new_protected:cpn { __color_export_comma-sep-cmyk:Nw }
  #1#2 ~ #3 ~ #4 ~ #5 \s__color_stop
  { \tl_set:Nn #1 { #2 , #3 , #4 , #5 } }
\cs_new_protected:cpn { __color_export_space-sep-cmyk:Nw } #1#2 \s__color_stop
  { \tl_set:Nn #1 {#2} }
```

(*End of definition for* `\__color_export_space-sep-cmyk:Nw` *and* `\__color_export_comma-sep-cmyk:Nw`.)

__color_export_comma-sep-rgb:Nw
__color_export_HTML:Nw
__color_export_space-sep-rgb:Nw
__color_export_HTML:n

HTML values must be given in `rgb`: we force conversion if required, then do some simple maths.

```
\cs_new_protected:cpn { __color_export_comma-sep-rgb:Nw } #1#2 ~ #3 ~ #4 \s__color_stop
  { \tl_set:Ne #1 { #2 , #3 , #4 } }
\cs_new_protected:Npn \__color_export_HTML:Nw #1#2 ~ #3 ~ #4 \s__color_stop
  {
    \tl_set:Ne #1
      {
        \__color_export_HTML:n {#2}
        \__color_export_HTML:n {#3}
        \__color_export_HTML:n {#4}
      }
  }
\cs_new:Npn \__color_export_HTML:n #1
  {
    \fp_compare:nNnTF {#1} = { 0 }
      { 00 }
      {
        \fp_compare:nNnT { #1 * 255 } < { 16 } { 0 }
        \int_to_Hex:n { \fp_to_int:n { #1 * 255 } }
      }
  }
\cs_new_protected:cpn { __color_export_space-sep-rgb:Nw } #1#2 \s__color_stop
  { \tl_set:Nn #1 {#2} }
```

(*End of definition for* `\__color_export_comma-sep-rgb:Nw` *and others.*)

93.12 Additional color models

\l__color_internal_prop

```
\prop_new:N \l__color_internal_prop
```

(*End of definition for* `\l__color_internal_prop`.)

\g__color_model_int

A tracker for the total number of new models.

```
\int_new:N \g__color_model_int
```

(*End of definition for* `\g__color_model_int`.)

\c__color_fallback_cmyk_tl
\c__color_fallback_gray_tl
\c__color_fallback_rgb_tl

For every colorspace, we define one of the base colorspaces as a fallback. The base colorspaces themselves are their own fallback.

```
\tl_const:Nn \c__color_fallback_cmyk_tl { cmyk }
\tl_const:Nn \c__color_fallback_gray_tl { gray }
\tl_const:Nn \c__color_fallback_rgb_tl { rgb }
```

(*End of definition for* `\c__color_fallback_cmyk_tl`*,* `\c__color_fallback_gray_tl`*, and* `\c__color_fallback_rgb_tl`*.*)

\g__color_colorants_prop

Mapping from names to colorants.

```
\prop_new:N \g__color_colorants_prop
\prop_gput:Nnn \g__color_colorants_prop { black }   { Black }
\prop_gput:Nnn \g__color_colorants_prop { blue }    { Blue }
\prop_gput:Nnn \g__color_colorants_prop { cyan }    { Cyan }
\prop_gput:Nnn \g__color_colorants_prop { green }   { Green }
\prop_gput:Nnn \g__color_colorants_prop { magenta } { Magenta }
\prop_gput:Nnn \g__color_colorants_prop { none }    { None }
\prop_gput:Nnn \g__color_colorants_prop { red }     { Red }
\prop_gput:Nnn \g__color_colorants_prop { yellow }  { Yellow }
```

(*End of definition for* `\g__color_colorants_prop`*.*)

\c__color_model_whitepoint_CIELAB_a_tl
\c__color_model_whitepoint_CIELAB_b_tl
\c__color_model_whitepoint_CIELAB_e_tl
\c__color_model_whitepoint_CIELAB_d50_tl
\c__color_model_whitepoint_CIELAB_d55_tl
\c__color_model_whitepoint_CIELAB_d65_tl
\c__color_model_whitepoint_CIELAB_d75_tl

Whitepoint data for the CIELAB profiles.

```
\tl_const:Nn \c__color_model_whitepoint_CIELAB_a_tl      { 1.0985 ~ 1 ~ 0.3558 }
\tl_const:Nn \c__color_model_whitepoint_CIELAB_b_tl      { 0.9807 ~ 1 ~ 1.1822 }
\tl_const:Nn \c__color_model_whitepoint_CIELAB_e_tl      { 1 ~ 1 ~ 1 }
\tl_const:cn { c__color_model_whitepoint_CIELAB_d50_tl } { 0.9642 ~ 1 ~ 0.8251 }
\tl_const:cn { c__color_model_whitepoint_CIELAB_d55_tl } { 0.9568 ~ 1 ~ 0.9214 }
\tl_const:cn { c__color_model_whitepoint_CIELAB_d65_tl } { 0.9504 ~ 1 ~ 1.0888 }
\tl_const:cn { c__color_model_whitepoint_CIELAB_d75_tl } { 0.9497 ~ 1 ~ 1.2261 }
```

(*End of definition for* `\c__color_model_whitepoint_CIELAB_a_tl` *and others.*)

\c__color_model_range_CIELAB_tl

The range for CIELAB color spaces.

```
\tl_const:Nn \c__color_model_range_CIELAB_tl { 0 ~ 100 ~ -128 ~ 127 ~ -128 ~ 127 }
```

(*End of definition for* `\c__color_model_range_CIELAB_tl`*.*)

\g__color_alternative_model_prop

For tracking the alternative model set up for separations, etc.

```
\prop_new:N \g__color_alternative_model_prop
\clist_map_inline:nn { cyan , magenta , yellow , black }
  { \prop_gput:Nnn \g__color_alternative_model_prop {#1} { cmyk } }
\clist_map_inline:nn { red , green , blue }
  { \prop_gput:Nnn \g__color_alternative_model_prop {#1} { rgb } }
```

(*End of definition for* `\g__color_alternative_model_prop`*.*)

\g__color_alternative_values_prop

Same for the values: a bit more involved.

```
\prop_new:N \g__color_alternative_values_prop
\prop_gput:Nnn \g__color_alternative_values_prop { cyan }    {  1 , 0 , 0 , 0 }
\prop_gput:Nnn \g__color_alternative_values_prop { magenta } {  0 , 1 , 0 , 0 }
\prop_gput:Nnn \g__color_alternative_values_prop { yellow }  {  0 , 0 , 1 , 0 }
\prop_gput:Nnn \g__color_alternative_values_prop { black }   {  0 , 0 , 0 , 1 }
\prop_gput:Nnn \g__color_alternative_values_prop { red }   {  1 , 0 , 0 }
\prop_gput:Nnn \g__color_alternative_values_prop { green } {  0 , 1 , 0 }
\prop_gput:Nnn \g__color_alternative_values_prop { blue }  {  0 , 0 , 1 }
```

(*End of definition for* `\g__color_alternative_values_prop`*.*)

\color_model_new:nnn
__color_model_new:nnn

Set up a new model: in general this has to be handled by a family-dependent function. To avoid some "interesting" questions with casing, we fold the case of the family name. The key–value list should always be present, so we convert it up-front to a `prop`, then deal with the detail on a per-family basis.

```
\cs_new_protected:Npn \color_model_new:nnn #1#2#3
  {
    \exp_args:Nee \__color_model_new:nnn
      { \tl_to_str:n {#1} }
      { \str_casefold:n {#2} } {#3}
  }
\cs_new_protected:Npn \__color_model_new:nnn #1#2#3
  {
    \cs_if_exist:cTF { __color_parse_model_ #1 :w }
      {
        \msg_error:nnn { color } { model-already-defined } {#1}
      }
      {
        \cs_if_exist:cTF { __color_model_ #2 :n }
          {
            \prop_set_from_keyval:Nn \l__color_internal_prop {#3}
            \use:c { __color_model_ #2 :n } {#1}
          }
          {
            \msg_error:nnn { color } { unknown-model-type } {#2}
          }
      }
  }
```

(*End of definition for* `\color_model_new:nnn` *and* `\__color_model_new:nnn`. *This function is documented on page 326.*)

__color_model_init:nnn
__color_model_init:nne

A shared auxiliary to do the basics of setting up a new model: reserve a number, create a white-equivalent, set up links to the backend.

```
\cs_new_protected:Npn \__color_model_init:nnn #1#2#3
  {
    \int_gincr:N \g__color_model_int
    \clist_map_inline:nn { fill , stroke , select }
      {
        \cs_new_protected:cpe { __color_backend_ ##1 _ #1 :n } ####1
          {
            \exp_not:c { __color_backend_ ##1 _ #2 :nn }
              { color \int_use:N \g__color_model_int } {####1}
          }
      }
    \cs_new_protected:cpe { __color_model_ #1 _white: }
      {
        \prop_put:Nnn \exp_not:N \l__color_named_white_prop {#1}
          { \exp_not:n {#3} }
        \exp_not:N \int_compare:nNnF { \tex_currentgrouplevel:D } = 0
          { \group_insert_after:N \exp_not:c { __color_model_ #1 _ white: } }
      }
    \use:c { __color_model_ #1 _white: }
  }
\cs_generate_variant:Nn \__color_model_init:nnn { nne }
```

(*End of definition for* `\__color_model_init:nnn`.)

__color_model_separation:n
__color_model_separation:nn
__color_model_separation:nnn
__color_model_separation:w
__color_model_separation_cmyk:nnnnnn
__color_model_separation_gray:nnnnnn
__color_model_separation_rgb:nnnnnn
__color_model_convert:nnn
__color_model_separation_CIELAB:nnnnnn
__color_model_separation_CIELAB:nnnnnnn

Separations must have a "real" name, which is pretty easy to find.

```
\cs_new_protected:Npn \__color_model_separation:n #1
  {
    \prop_get:NnNTF \l__color_internal_prop { name }
      \l__color_internal_tl
      {
        \exp_args:NV \__color_model_separation:nn
          \l__color_internal_tl {#1}
      }
      {
        \msg_error:nnn { color }
          { separation-requires-name } {#1}
      }
  }
```

We have two keys to find at this stage: the alternative space model and linked values.

```
\cs_new_protected:Npn \__color_model_separation:nn #1#2
  {
    \prop_get:NnNTF \l__color_internal_prop { alternative-model }
      \l__color_internal_tl
      {
        \exp_args:NV \__color_model_separation:nnn
          \l__color_internal_tl {#2} {#1}
      }
      {
        \msg_error:nnn { color }
          { separation-alternative-model } {#2}
      }
  }
\cs_new_protected:Npn \__color_model_separation:nnn #1#2#3
  {
    \cs_if_exist:cTF { __color_model_separation_ #1 :nnnnnn }
      {
        \prop_get:NnNTF \l__color_internal_prop { alternative-values }
          \l__color_internal_tl
          {
            \exp_after:wN \__color_model_separation:w \l__color_internal_tl
              , 0 , 0 , 0 , 0 \s__color_stop {#2} {#3} {#1}
          }
          {
            \msg_error:nnn { color }
              { separation-alternative-values } {#2}
          }
      }
      {
        \msg_error:nnn { color }
          { unknown-alternative-model } {#1}
      }
  }
```

As each alternative space leads to a different requirement for conversion, and as there are only a small number of choices, we manually split the data and then set up. Notice that mixing tints is really just the same as mixing `gray`. The `white` color is special, as it allows tints to be adjusted without an additional color space. To make sure the data is set for that at all group levels, we need to work on a per-level basis. Within the output, only the set-up needs the "real" name of the colorspace: we use a simple tracking number for general usage as this is a clear namespace without issues of escaping chars.

```
\cs_new_protected:Npn \__color_model_separation:w
  #1 , #2 , #3 , #4 , #5 \s__color_stop #6#7#8
  {
    \__color_model_init:nnn {#6} { separation } { 0 }
    \cs_new_eq:cN { __color_parse_mix_ #6 :nw } \__color_parse_mix_gray:nw
    \cs_new:cpn { __color_parse_model_ #6 :w } ##1 , ##2 \s__color_stop
      { {#6} { \__color_parse_number:n {##1} } }
    \use:c { __color_model_separation_ #8 :nnnnnn }
      {#6} {#7} {#1} {#2} {#3} {#4}
    \prop_gput:Nnn \g__color_alternative_model_prop {#6} {#8}
    \prop_gput:Nne \g__color_colorants_prop {#6}
      { \str_convert_pdfname:n {#7} }
  }
\cs_new_protected:Npn \__color_model_separation_cmyk:nnnnnn #1#2#3#4#5#6
  {
    \tl_const:cn { c__color_fallback_ #1 _tl } { cmyk }
    \cs_new:cpn { __color_convert_ #1 _cmyk:w } ##1 \s__color_stop
      {
        \fp_eval:n {##1 * #3} ~
        \fp_eval:n {##1 * #4} ~
        \fp_eval:n {##1 * #5} ~
        \fp_eval:n {##1 * #6}
      }
    \cs_new:cpn { __color_convert_cmyk_ #1 :w } ##1 \s__color_stop { 1 }
    \prop_gput:Nnn \g__color_alternative_values_prop {#1} { #3 , #4 , #5 , #6 }
    \__color_backend_separation_init:nnnnn {#2} { /DeviceCMYK } { }
      { 0 ~ 0 ~ 0 ~ 0 } { #3 ~ #4 ~ #5 ~ #6 }
  }
\cs_new_protected:Npn \__color_model_separation_rgb:nnnnnn #1#2#3#4#5#6
  {
    \tl_const:cn { c__color_fallback_ #1 _tl } { rgb }
    \cs_new:cpn { __color_convert_ #1 _rgb:w } ##1 \s__color_stop
      {
        \fp_eval:n {##1 * #3} ~
        \fp_eval:n {##1 * #4} ~
        \fp_eval:n {##1 * #5}
      }
    \cs_new:cpn { __color_convert_rgb_ #1 :w } ##1 \s__color_stop { 1 }
    \prop_gput:Nnn \g__color_alternative_values_prop {#1} { #3 , #4 , #5 }
    \__color_backend_separation_init:nnnnn {#2} { /DeviceRGB } { }
      { 0 ~ 0 ~ 0 } { #3 ~ #4 ~ #5 }
  }
\cs_new_protected:Npn \__color_model_separation_gray:nnnnnn #1#2#3#4#5#6
  {
    \tl_const:cn { c__color_fallback_ #1 _tl } { gray }
    \cs_new:cpn { __color_convert_ #1 _gray:w } ##1 \s__color_stop
      { \fp_eval:n {##1 * #3} }
    \cs_new:cpn { __color_convert_gray_ #1 :w } ##1 \s__color_stop { 1 }
    \prop_gput:Nnn \g__color_alternative_values_prop {#1} {#3}
    \__color_backend_separation_init:nnnnn {#2} { /DeviceGray } { } { 0 } {#3}
```

```
  }
```

Generic model conversion *via* an alternative intermediate.

```
\cs_new_protected:Npn \__color_model_convert:nnn #1#2#3
  {
    \cs_new:cpe { __color_convert_ #1 _ #3 :w } ##1 \s__color_stop
      {
        \exp_not:N \exp_args:NNe \exp_not:N \use:nn
        \exp_not:c { __color_convert_  #2 _ #3 :w }
          { \exp_not:c { __color_convert_ #1 _ #2 :w } ##1 \s__color_stop }
          \c_space_tl \exp_not:N \s__color_stop
      }
  }
```

Setting up for CIELAB needs a bit more work: there is the illuminant and the need for an appropriate object.

```
\cs_new_protected:Npn \__color_model_separation_CIELAB:nnnnnn #1#2#3#4#5#6
  {
    \prop_get:NnNF \l__color_internal_prop { illuminant }
      \l__color_internal_tl
      {
        \msg_error:nnn { color }
          { CIELAB-requires-illuminant } {#1}
        \tl_set:Nn \l__color_internal_tl { d50 }
      }
    \exp_args:NV \__color_model_separation_CIELAB:nnnnnnn
      \l__color_internal_tl {#1} {#2} {#3} {#4} {#5} {#6}
  }
```

If a CIELAB space is being set up, we need the illuminant, then create the appropriate set up. At present, this doesn't include `BlackPoint` or `Range` data, but that may be added later. As CIELAB colors cannot be converted to anything else, we fallback to producing black in the gray colorspace: the user should set up a second model for colors set up this way.

```
\cs_new_protected:Npn \__color_model_separation_CIELAB:nnnnnnn #1#2#3#4#5#6#7
  {
    \tl_if_exist:cTF { c__color_model_whitepoint_CIELAB_ #1 _tl }
      {
        \__color_backend_separation_init_CIELAB:nnn {#1} {#3} { #4 ~ #5 ~ #6 }
        \tl_const:cn { c__color_fallback_ #2 _tl } { gray }
        \cs_new:cpn { __color_convert_ #2 _gray:w } ##1 \s__color_stop
          { 0 }
        \cs_new:cpn { __color_convert_gray_ #2 :w } ##1 \s__color_stop
          { 1 }
      }
      {
        \msg_error:nnn { color }
          { unknown-CIELAB-illuminant } {#1}
      }
  }
```

(*End of definition for* `\__color_model_separation:n` *and others.*)

__color_model_devicen:n
__color_model_devicen:nn
__color_model_devicen:nnn
__color_model_devicen:nnnn
__color_model_devicen_parse_1:nn
__color_model_devicen_parse_2:nn
__color_model_devicen_parse_3:nn
__color_model_devicen_parse_4:nn
__color_model_devicen_parse_generic:nn
__color_model_devicen_parse:nw
__color_model_devicen_mix:nw
__color_model_devicen_init:nnn

We require a list of component names here: one might call them colorants, but it's convenient to use TEX names instead so we slightly adjust the terminology.

```
\cs_new_protected:Npn \__color_model_devicen:n #1
  {
    \prop_get:NnNTF \l__color_internal_prop { names }
      \l__color_internal_tl
      {
        \exp_args:NV \__color_model_devicen:nn
          \l__color_internal_tl {#1}
      }
      {
        \msg_error:nnn { color }
          { DeviceN-requires-names } {#1}
      }
  }
```

All valid models will have an alternative listed, either hard-coded for the core device ones, or dynamically added for Separations, etc.

```
\cs_new_protected:Npn \__color_model_devicen:nn #1#2
  {
    \tl_clear:N \l__color_model_tl
    \clist_map_inline:nn {#1}
      {
        \prop_get:NnNTF \g__color_alternative_model_prop {##1}
          \l__color_internal_tl
          {
            \tl_if_empty:NTF \l__color_model_tl
              { \tl_set_eq:NN \l__color_model_tl \l__color_internal_tl }
              {
                \str_if_eq:VVF \l__color_model_tl \l__color_internal_tl
                  {
                    \msg_error:nnn { color }
                      { DeviceN-inconsistent-alternative }
                      {#2}
                    \clist_map_break:n { \use_none:nnnn }
                  }
              }
          }
          {
            \str_if_eq:nnF {##1} { none }
              {
                \msg_error:nnn { color }
                  { DeviceN-no-alternative }
                  {#2}
              }
          }
      }
    \tl_if_empty:NTF \l__color_model_tl
      {
        \msg_error:nnn { color }
          { DeviceN-no-alternative } {#2}
      }
      { \exp_args:NV \__color_model_devicen:nnn \l__color_model_tl {#1} {#2} }
  }
```

We now complete the data we require by first finding out how many colorants there are, then moving on to begin constructing the function required to map to the alternative color space.

```
\cs_new_protected:Npn \__color_model_devicen:nnn #1#2#3
  {
    \exp_args:Ne \__color_model_devicen:nnnn
      { \clist_count:n {#2} } {#1} {#2} {#3}
  }
```

At this stage, we have checked everything is in place, so we can set up the TEX and backend data structures. As for separations, it's not really possible in general to have a fallback, so we simply provide "black" for each element.

```
\cs_new_protected:Npn \__color_model_devicen:nnnn #1#2#3#4
  {
    \__color_model_init:nne {#4} { devicen }
      {
        0 \prg_replicate:nn { #1 - 1 } { ~ 0 }
      }
    \cs_if_exist_use:cF { __color_model_devicen_parse_ #1 :nn }
      { \__color_model_devicen_parse_generic:nn }
        {#4} {#1}
    \__color_model_devicen_init:nnn {#1} {#2} {#3}
    \__color_model_devicen_convert:nnne {#4} {#2} {#3}
      {
        1 \prg_replicate:nn { #1 - 1 } { ~ 1 }
      }
  }
```

For short lists of DeviceN colors, we can use hand-tuned parsing. This lines up with other models, where we allow for up to four components. For larger spaces, rather than limit artificially, we use a somewhat slow approach based on open-ended commas-lists.

```
\cs_new_protected:cpn { __color_model_devicen_parse_1:nn } #1#2
  {
    \cs_new:cpn { __color_parse_model_ #1 :w  } ##1 , ##2 \s__color_stop
      { {#1} { \__color_parse_number:n {##1} } }
    \cs_new_eq:cN { __color_parse_mix_ #1 :nw  } \__color_parse_mix_gray:nw
  }
\cs_new_protected:cpn { __color_model_devicen_parse_2:nn } #1#2
  {
    \cs_new:cpn { __color_parse_model_ #1 :w  } ##1 , ##2 , ##3 \s__color_stop
      { {#1} { \__color_parse_number:n {##1} ~ \__color_parse_number:n {##2} } }
    \cs_new:cpn { __color_parse_mix_ #1 :nw }
      ##1##2 ~ ##3 \s__color_mark ##4 ~ ##5 \s__color_stop
      {
        \fp_eval:n { ##2 * ##1 + ##4 * ( 1 - ##1 ) } \c_space_tl
        \fp_eval:n { ##3 * ##1 + ##5 * ( 1 - ##1 ) }
      }
  }
\cs_new_protected:cpn { __color_model_devicen_parse_3:nn } #1#2
  {
    \cs_new:cpn { __color_parse_model_ #1 :w  } ##1 , ##2 , ##3 , ##4 \s__color_stop
      {
        {#1}
        {
          \__color_parse_number:n {##1} ~
          \__color_parse_number:n {##2} ~
```

```
          \__color_parse_number:n {##3}
        }
      }
    \cs_new_eq:cN { __color_parse_mix_ #1 :nw  } \__color_parse_mix_rgb:nw
  }
\cs_new_protected:cpn { __color_model_devicen_parse_4:nn } #1#2
  {
    \cs_new:cpn { __color_parse_model_ #1 :w  }
      ##1 , ##2 , ##3 , ##4 , ##5 \s__color_stop
      {
        {#1}
        {
          \__color_parse_number:n {##1} ~
          \__color_parse_number:n {##2} ~
          \__color_parse_number:n {##3} ~
          \__color_parse_number:n {##4}
        }
      }
  \cs_new_eq:cN { __color_parse_mix_ #1 :nw } \__color_parse_mix_cmyk:nw
  }
\cs_new_protected:Npn \__color_model_devicen_parse_generic:nn #1#2
  {
    \cs_new:cpn { __color_parse_model_ #1 :w  } ##1 , ##2 \s__color_stop
      {
        {#1}
        { \__color_model_devicen_parse:nw {#2} ##1 , ##2 , \q_nil , \s__color_stop }
      }
    \cs_new:cpe { __color_parse_mix_ #1 :nw }
      ##1 ##2 \s__color_mark ##3 \s__color_stop
      {
        \exp_not:N \__color_model_devicen_mix:nw {##1}
          ##2 \c_space_tl \exp_not:N \q_nil \c_space_tl \exp_not:N \s__color_mark
          ##3 \c_space_tl \exp_not:N \q_nil \c_space_tl \exp_not:N \s__color_stop
      }
  }
\cs_new:Npn \__color_model_devicen_parse:nw #1#2 , #3 \s__color_stop
  {
    \int_compare:nNnT {#1} > 0
      {
        \quark_if_nil:nTF {#2}
          { \prg_replicate:nn {#1} { 0 ~ } }
          {
            \__color_parse_number:n {#2}
            \int_compare:nNnT {#1} > 1 { ~ }
            \exp_args:Nf \__color_model_devicen_parse:nw
              { \int_eval:n { #1 - 1 } } #3 \s__color_stop
          }
      }
  }
\cs_new:Npn \__color_model_devicen_mix:nw #1#2 ~ #3 \s__color_mark #4 ~ #5 \s__color_stop
  {
    \fp_eval:n { #2 * #1 + #4 * ( 1 - #1 ) }
    \quark_if_nil:oF { \tl_head:w #3 \q_stop }
      {
```

```
    \c_space_tl
    \__color_model_devicen_mix:nw {#1} #3 \s__color_mark #5 \s__color_stop
   }
 }
```

To construct the tint transformation, we have to use PostScript. The aim is to have the final tint for each device colorant as

$$1 - \prod_n (1 - X_n D_{X_n})$$

where X is a DeviceN colorant and D is the amount of device colorant that the DeviceN colorant maps to. At the start of the process, the PostScript stack will contain the X_n values, whilst we have the D values on a per-DeviceN colorant basis. The more convenient approach for us is therefore to take each DeviceN colorant in turn and find the value $1 - X_n D_{X_n}$, multiplying as we go, and finalise with the subtraction. That contrasts to colorspace: it splits the process up by process color, which works better when you have a fixed list of colorants. (colorspace only supports up to 4 DeviceN colors, and only `cmyk` as the alternative space.) To set this up, we first need to know the number of values in the target color space: this is easily handled as there are a very small range of possibles. Once we have that information, it's relatively easy to build the required PostScript using some generic code.

```
\cs_new_protected:Npn \__color_model_devicen_init:nnn #1#2#3
 {
  \exp_args:Ne \__color_model_devicen_init:nnnn
   {
    \str_case:nn {#2}
     {
      { cmyk } { 4 }
      { gray } { 1 }
      { rgb }  { 3 }
     }
   }
   {#1} {#2} {#3}
 }
```

As we always need to split the alternative values into parts, we use a shared auxiliary and only use a minimal difference between code paths. Construction of the tint transformation is as far as possible done using loops, which means there are some inefficiencies for device colors in the `DeviceN` space: we roll the stack one-at-a-time even if there is a potential shortcut. However, that way there is nothing to special-case. Once this is sorted, we can write the tint transform object, which will remain as the last object until we sort out the final step: the colorant list.

```
\cs_new_protected:Npn \__color_model_devicen_init:nnnn #1#2#3#4
 {
  \tl_set:Ne \l__color_internal_tl
   { \prg_replicate:nn {#1} { 1.0 ~ } }
  \int_zero:N \l__color_internal_int
  \clist_map_inline:nn {#4}
   {
    \int_incr:N \l__color_internal_int
    \prop_get:NnN \g__color_alternative_values_prop {##1}
     \l__color_value_tl
    \exp_after:wN \__color_model_devicen_transform:w
```

```
        \l__color_value_tl , 0 , 0 , 0 , \s__color_stop {#1} {#2}
      }
    \tl_put_right:Ne \l__color_internal_tl
      {
        \prg_replicate:nn {#1}
          { neg ~ 1.0 ~ add ~ #1 ~ -1 ~ roll ~ }
        \int_eval:n { #2 + #1 } ~ #1 ~ roll
        \prg_replicate:nn {#2} { ~ pop } ~
        #1 ~ 1 ~ roll
      }
    \use:e
      {
        \__color_backend_devicen_init:nnn
          {
            \clist_map_function:nN {#4}
              \__color_model_devicen_colorant:n
          }
          {
            \str_case:nn {#3}
              {
                { cmyk } { /DeviceCMYK }
                { gray } { /DeviceGray }
                { rgb }  { /DeviceRGB }
              }
          }
          { \exp_not:V \l__color_internal_tl }
      }
  }
\cs_new_protected:Npn \__color_model_devicen_transform:w
  #1 , #2 , #3 , #4 , #5 \s__color_stop #6#7
  {
    \use:c { __color_model_devicen_transform_ #6 :nnnnn }
      {#1} {#2} {#3} {#4} {#7}
  }
\cs_new_protected:cpn { __color_model_devicen_transform_1:nnnnn } #1#2#3#4#5
  { \__color_model_devicen_transform:nnn {#5} { 1 } {#1} }
\cs_new_protected:cpn { __color_model_devicen_transform_3:nnnnn } #1#2#3#4#5
  {
    \clist_map_inline:nn { #1 , #2 , #3 }
      { \__color_model_devicen_transform:nnn {#5} { 3 } {##1} }
  }
\cs_new_protected:cpn { __color_model_devicen_transform_4:nnnnn } #1#2#3#4#5
  {
    \clist_map_inline:nn { #1 , #2 , #3 , #4 }
      { \__color_model_devicen_transform:nnn {#5} { 4 } {##1} }
  }
\cs_new_protected:Npn \__color_model_devicen_transform:nnn #1#2#3
  {
    \tl_put_right:Ne \l__color_internal_tl
      {
        \fp_compare:nNnF {#3} = \c_zero_fp
          {
            \int_eval:n { #1 - \l__color_internal_int + #2 } ~ index ~
              -#3 ~ mul ~ 1.0 ~ add ~ mul ~
```

```
     }
    #2 ~ -1 ~ roll ~
   }
 }
\cs_new:Npn \__color_model_devicen_colorant:n #1
 {
  / \prop_item:Nn \g__color_colorants_prop {#1} ~
 }
```

Here we need to set up conversion from the DeviceN space to the alternative at the TEX level. This also means supplying methods for inter-converting to other parameter-based spaces. Essentially the approach is exactly the same as the PostScript, just expressed in TEX terms.

```
\cs_new_protected:Npn \__color_model_devicen_convert:nnnn #1#2#3
 {
  \use:c { __color_model_devicen_convert_ #2 :nnn } {#1} {#3}
 }
\cs_generate_variant:Nn \__color_model_devicen_convert:nnnn { nnne }
\cs_new_protected:Npn \__color_model_devicen_convert_cmyk:nnn #1#2
 {
  \tl_const:cn { c__color_fallback_ #1 _tl } { cmyk }
  \__color_model_devicen_convert:nnnnn {#1} { cmyk } { 4 } {#2}
 }
\cs_new_protected:Npn \__color_model_devicen_convert_gray:nnn #1#2
 {
  \tl_const:cn { c__color_fallback_ #1 _tl } { gray }
  \__color_model_devicen_convert:nnnnn {#1} { gray } { 1 } {#2}
 }
\cs_new_protected:Npn \__color_model_devicen_convert_rgb:nnn #1#2
 {
  \tl_const:cn { c__color_fallback_ #1 _tl } { rgb }
  \__color_model_devicen_convert:nnnnn {#1} { rgb } { 3 } {#2}
 }
\cs_new_protected:Npn \__color_model_devicen_convert:nnnnn #1#2#3#4#5
 {
  \cs_new:cpn { __color_convert_ #2 _ #1 :w } ##1 \s__color_stop {#5}
  \cs_new:cpe { __color_convert_ #1 _ #2 :w } ##1 \s__color_stop
   {
    \exp_not:c { __color_convert_devicen_ #2 : \prg_replicate:nn {#3} { n } w }
     \prg_replicate:nn {#3} { { 1 } }
     ##1 ~ \exp_not:N \s__color_mark
     \clist_map_function:nN {#4} \__color_model_devicen_convert:n
     {}
     \exp_not:N \s__color_stop
   }
 }
\cs_new:Npn \__color_model_devicen_convert:n #1
 {
  {
   \exp_args:Ne \__color_model_devicen_convert_aux:n
    { \prop_item:Nn \g__color_alternative_values_prop {#1} }
  }
 }
\cs_new:Npn \__color_model_devicen_convert_aux:n #1
```

```
  { \__color_model_devicen_convert_aux:w #1 , , , , \s__color_stop }
\cs_new:Npn \__color_model_devicen_convert_aux:w #1 , #2 , #3 , #4 , #5 \s__color_stop
  {
    {#1}
    \tl_if_blank:nF {#2}
      {
        {#2}
        \tl_if_blank:nF {#3}
          {
            {#3}
            \tl_if_blank:nF {#4} { {#4} }
          }
      }
  }
\cs_new:Npn \__color_convert_devicen_cmyk:nnnnw
  #1#2#3#4#5 ~ #6 \s__color_mark #7#8 \s__color_stop
  {
    \__color_convert_devicen_cmyk:nnnnnnnnn {#5} {#1} {#2} {#3} {#4} #7
      #6 \s__color_mark #8 \s__color_stop
  }
\cs_new:Npn \__color_convert_devicen_cmyk:nnnnnnnnn #1#2#3#4#5#6#7#8#9
  {
    \use:e
      {
        \exp_not:N \__color_convert_devicen_cmyk_aux:nnnnw
          { \fp_eval:n { #2 * (1 - (#1 * #6)) } }
          { \fp_eval:n { #3 * (1 - (#1 * #7)) } }
          { \fp_eval:n { #4 * (1 - (#1 * #8)) } }
          { \fp_eval:n { #5 * (1 - (#1 * #9)) } }
      }
  }
\cs_new:Npn \__color_convert_devicen_cmyk_aux:nnnnw
  #1#2#3#4 #5 \s__color_mark #6 \s__color_stop
  {
    \tl_if_blank:nTF {#5}
      {
        \fp_eval:n { 1 - #1 } ~
        \fp_eval:n { 1 - #2 } ~
        \fp_eval:n { 1 - #3 } ~
        \fp_eval:n { 1 - #4 }
      }
      {
        \__color_convert_devicen_cmyk:nnnnw {#1} {#2} {#3} {#4}
          #5 \s__color_mark #6 \s__color_stop
      }
  }
\cs_new:Npn \__color_convert_devicen_gray:nw
  #1#2 ~ #3 \s__color_mark #4#5 \s__color_stop
  {
    \__color_convert_devicen_gray:nnn {#2} {#1} #4
      #3 \s__color_mark #5 \s__color_stop
  }
\cs_new:Npn \__color_convert_devicen_gray:nnn #1#2#3
  {
```

```
    \exp_arsgs:Ne \__color_convert_devicen_gray_aux:nw
      { \fp_eval:n { #2 * (1 - (#1 * #3)) } }
  }
\cs_new:Npn \__color_convert_devicen_gray_aux:nw
  #1 #2 \s__color_mark #3 \s__color_stop
  {
    \tl_if_blank:nTF {#2}
      { \fp_eval:n { 1 - #1 } }
      {
        \__color_convert_devicen_gray:nw {#1}
          #2 \s__color_mark #3 \s__color_stop
      }
  }
\cs_new:Npn \__color_convert_devicen_rgb:nnnw
  #1#2#3#4 ~ #5 \s__color_mark #6#7 \s__color_stop
  {
    \__color_convert_devicen_rgb:nnnnnnn {#4} {#1} {#2} {#3} #6
      #5 \s__color_mark #7 \s__color_stop
  }
\cs_new:Npn \__color_convert_devicen_rgb:nnnnnnn #1#2#3#4#5#6#7
  {
    \use:e
      {
        \exp_not:N \__color_convert_devicen_rgb_aux:nnnw
          { \fp_eval:n { #2 * (1 - (#1 * #5)) } }
          { \fp_eval:n { #3 * (1 - (#1 * #6)) } }
          { \fp_eval:n { #4 * (1 - (#1 * #7)) } }
      }
  }
\cs_new:Npn \__color_convert_devicen_rgb_aux:nnnw
  #1#2#3 #4 \s__color_mark #5 \s__color_stop
  {
    \tl_if_blank:nTF {#4}
      {
        \fp_eval:n { 1 - #1 } ~
        \fp_eval:n { 1 - #2 } ~
        \fp_eval:n { 1 - #3 }
      }
      {
        \__color_convert_devicen_rgb:nnnw {#1} {#2} {#3}
          #4 \s__color_mark #5 \s__color_stop
      }
  }
```

(*End of definition for* `\__color_model_devicen:n` *and others.*)

\c__color_icc_colorspace_signatures_prop

The signatures in the ICC file header indicating the underlying colorspace. We map it to three values: The number of components, the values corresponding to white, and the range.

```
\prop_const_from_keyval:Nn \c__color_icc_colorspace_signatures_prop
  {
% Gray
    47524159 = {1} {1} {0} {},
% RGB
```

```
    52474220 = {3} {0~0~0} {1~1~1} {},
% CMYK
    434D594B = {4} {0~0~0~1} {0~0~0~0} {},
% Lab
    4C616220 = {3} {0~0~0} {100~0~0} {0~100~-128~127~-128~127}
  }
```

(*End of definition for* `\c__color_icc_colorspace_signatures_prop`.)

`\__color_model_iccbased:n`
`\__color_model_iccbased:nn`
`\__color_model_iccbased:nnn`
`\__color_model_iccbased_aux:nnn`

For an ICC profile, we need a file name and a number of components. The file name is processed here so the backend can treat it as a string.

```
\cs_new_protected:Npn \__color_model_iccbased:n #1
  {
    \prop_get:NnNTF \l__color_internal_prop { file }
      \l__color_internal_tl
      {
        \exp_args:NV \__color_model_iccbased:nn
          \l__color_internal_tl {#1}
      }
      {
        \msg_error:nnn { color }
          { ICCBased-requires-file } {#1}
      }
  }
\cs_new_protected:Npn \__color_model_iccbased:nn #1#2
  {
    \prop_get:NeNTF \c__color_icc_colorspace_signatures_prop
      { \file_hex_dump:nnn { #1 } { 17 } { 20 } } \l__color_internal_tl
      {
        \exp_last_unbraced:NV \__color_model_iccbased_aux:nnnnnn
          \l__color_internal_tl { #2 } { #1 }
      }
      {
        \msg_error:nnn { color }
        { ICCBased-unsupported-colorspace } {#2}
      }
  }
```

Here, we can use the same internals as for DeviceN approach as we know the number of components. No conversion is possible, so there is no need to worry about that at all.

```
\cs_new_protected:Npn \__color_model_iccbased_aux:nnnnnn #1#2#3#4#5#6
  {
    \__color_model_init:nnn {#5} { iccbased } {#3}
    \tl_const:cn { c__color_fallback_ #5 _tl } { gray }
    \cs_new:cpn { __color_convert_ #5 _gray:w } ##1 \s__color_stop { 0 }
    \cs_new:cpn { __color_convert_gray_ #5 :w } ##1 \s__color_stop { #2 }
    \use:c { __color_model_devicen_parse_ #1 :nn } {#5} {#1}
    \exp_args:Ne \__color_backend_iccbased_init:nnn
      { \file_full_name:n {#6} } {#1} {#4}
  }
```

(*End of definition for* `\__color_model_iccbased:n` *and others.*)

93.13 Applying profiles

\color_profile_apply:nn
__color_profile_apply:nn
__color_profile_apply_gray:n
__color_profile_apply_rgb:n
__color_profile_apply_cmyk:n

With a limited range of outcomes, this is largely about getting data to the backend.

```
\cs_new_protected:Npn \color_profile_apply:nn #1#2
  {
    \exp_args:Ne \__color_profile_apply:nn
      { \file_full_name:n {#1} } {#2}
  }
\cs_new_protected:Npn \__color_profile_apply:nn #1#2
  {
    \cs_if_exist_use:cF { __color_profile_apply_ \tl_to_str:n {#2} :n }
      {
        \msg_error:nnn { color } { ICC-Device-unknown } {#2}
        \use_none:n
      }
        {#1}
  }
\cs_new_protected:Npn \__color_profile_apply_gray:n #1
  {
    \int_gincr:N \g__color_model_int
    \__color_backend_iccbased_device:nnn {#1} { Gray } { 1 }
  }
\cs_new_protected:Npn \__color_profile_apply_rgb:n #1
  {
    \int_gincr:N \g__color_model_int
    \__color_backend_iccbased_device:nnn {#1} { RGB } { 3 }
  }
\cs_new_protected:Npn \__color_profile_apply_cmyk:n #1
  {
    \int_gincr:N \g__color_model_int
    \__color_backend_iccbased_device:nnn {#1} { CMYK } { 4 }
  }
```

(*End of definition for* `\color_profile_apply:nn` *and others. This function is documented on page 327.*)

93.14 Diagnostics

\color_show:n
\color_log:n
__color_show:Nn
__color_show:n

Extract the information about a color and format for the user: the approach is similar to the keys module here.

```
\cs_new_protected:Npn \color_show:n
  { \__color_show:Nn \msg_show:nneeee }
\cs_new_protected:Npn \color_log:n
  { \__color_show:Nn \msg_log:nneeee }
\cs_new_protected:Npn \__color_show:Nn #1#2
  {
    #1 { color } { show }
      {#2}
      {
        \color_if_exist:nT {#2}
          {
            \exp_args:Nv \__color_show:n { l__color_named_ #2 _tl }
            \prop_map_function:cN
              { l__color_named_ #2 _prop }
```

```
                \msg_show_item_unbraced:nn
            }
        }
        { }
        { }
    }
\cs_new:Npn \__color_show:n #1
  {
    \msg_show_item_unbraced:nn { model } {#1}
  }
```

(*End of definition for* `\color_show:n` *and others. These functions are documented on page 323.*)

93.15 Messages

```
\msg_new:nnnn { color } { CIELAB-requires-illuminant }
  { CIELAB~color~space~'#1'~require~an~illuminant. }
  {
    LaTeX~has~been~asked~to~create~a~separation~color~space~using~
    CIELAB~specifications,~but~no~\\ \\
    \iow_indent:n { illuminant~=~<basis> }
    \\ \\
    key~was~given~with~the~correct~information.~LaTeX~will~use~illuminant~
    'd50'~for~recovery.
  }
\msg_new:nnnn { color } { conversion-not-available }
  { No~model~conversion~available~from~'#1'~to~'#2'. }
  {
    LaTeX~has~been~asked~to~convert~a~color~from~model~'#1'~
    to~model'#2',~but~there~is~no~method~available~to~do~that.
  }
\msg_new:nnnn { color } { DeviceN-inconsistent-alternative }
  { DeviceN~color~spaces~require~a~single~alternative~space. }
  {
    LaTeX~has~been~asked~to~create~a~DeviceN~color~space~'#1',~
    but~the~constituent~colors~do~not~have~a~common~alternative~
    color.
  }
\msg_new:nnnn { color } { DeviceN-no-alternative }
  { DeviceN~color~spaces~require~an~alternative~space. }
  {
    LaTeX~has~been~asked~to~create~a~DeviceN~color~space~'#1',~
    but~the~constituent~colors~do~not~all~have~a~device-based~alternative.
  }
\msg_new:nnnn { color } { DeviceN-requires-names }
  { DeviceN~color~space~'#1'~require~a~list~of~names. }
  {
    LaTeX~has~been~asked~to~create~a~DeviceN~color~space,~
    but~no~\\ \\
    \iow_indent:n { names~=~<names> }
    \\ \\
    key~was~given~with~the~correct~information.
  }
\msg_new:nnnn { color } { ICC-Device-unknown }
```

```
  { Unknown~device~color~space~'#1'. }
  {
    LaTeX~has~been~asked~to~apply~an~ICC~profile~but~the~device~color~space~
    '#1'~is~unknown.
  }
\msg_new:nnnn { color } { ICCBased-unsupported-colorspace }
  { ICCBased~color~space~'#1'~uses~an~unsupported~data~color~space. }
  {
    LaTeX~has~been~asked~to~create~a~ICCBased~colorspace,~but~the~
    used~data~colorspace~is~not~supported.~ICC~profiles~used~for~
    defining~a~ICCBased~colorspace~should~use~a~Lab,~RGB,~or~
    CMYK~data~colorspace.~LaTeX~will~ignore~this~request.
  }
\msg_new:nnnn { color } { ICCBased-requires-file }
  { ICCBased~color~space~'#1'~require~an~file. }
  {
    LaTeX~has~been~asked~to~create~an~ICCBased~color~space,~but~no~\\ \\
    \iow_indent:n { file~=~<name> }
    \\ \\
    key~was~given~with~the~correct~information.~LaTeX~will~ignore~this~
    request.
  }
\msg_new:nnnn { color } { model-already-defined }
  { Color~model~'#1'~already~defined. }
  {
    LaTeX~was~asked~to~define~a~new~color~model~called~'#1',~but~
    this~color~model~already~exists.
  }
\msg_new:nnnn { color } { out-of-range }
  { Input~value~#1~out~of~range~[#2,~#3]. }
  {
    LaTeX~was~expecting~a~value~in~the~range~[#2,~#3]~as~part~of~a~color,~
    but~you~gave~#1.~LaTeX~will~assume~you~meant~the~limit~of~the~range~
    and~continue.
  }
\msg_new:nnnn { color } { separation-alternative-model }
  { Separation~color~space~'#1'~require~an~alternative~model. }
  {
    LaTeX~has~been~asked~to~create~a~separation~color~space,~
    but~no~\\ \\
    \iow_indent:n { alternative-model~=~<model> }
    \\ \\
    key~was~given~with~the~correct~information.
  }
\msg_new:nnnn { color } { separation-alternative-values }
  { Separation~color~space~'#1'~require~values~for~the~alternative~space. }
  {
    LaTeX~has~been~asked~to~create~a~separation~color~space,~
    but~no~\\ \\
    \iow_indent:n { alternative-values~=~<model> }
    \\ \\
    key~was~given~with~the~correct~information.
  }
\msg_new:nnnn { color } { separation-requires-name }
```

```
  { Separation~color~space~'#1'~require~a~formal~name. }
  {
    LaTeX~has~been~asked~to~create~a~separation~color~space,~
    but~no~\\ \\
    \iow_indent:n { name~=~<formal~name> }
    \\ \\
    key~was~given~with~the~correct~information.
  }
\msg_new:nnn { color } { unhandled-model }
  {
    Unhandled~color~model~in~LaTeX2e~value~"#1":
    \\ \\
    falling~back~on~grayscale.
  }
\msg_new:nnnn { color } { unknown-color }
  { Unknown~color~'#1'. }
  {
    LaTeX~has~been~asked~to~use~a~color~named~'#1',~
    but~this~has~never~been~defined.
  }
\msg_new:nnnn { color } { unknown-alternative-model }
  { Separation~color~space~'#1'~require~an~valid~alternative~space. }
  {
    LaTeX~has~been~asked~to~create~a~separation~color~space,~
    but~the~model~given~as\\ \\
    \iow_indent:n { alternative-model~=~<model> }
    \\ \\
    is~unknown.
  }
\msg_new:nnnn { color } { unknown-export-format }
  { Unknown~export~format~'#1'. }
  {
    LaTeX~has~been~asked~to~export~a~color~in~format~'#1',~
    but~this~has~never~been~defined.
  }
\msg_new:nnnn { color } { unknown-CIELAB-illuminant }
  { Unknown~illuminant~model~'#1'. }
  {
    LaTeX~has~been~asked~to~use~create~a~color~space~using~CIELAB~
    illuminant~'#1',~but~this~does~not~exist.
  }
\msg_new:nnnn { color } { unknown-model }
  { Unknown~color~model~'#1'. }
  {
    LaTeX~has~been~asked~to~use~a~color~model~called~'#1',~
    but~this~model~is~not~set~up.
  }
\msg_new:nnnn { color } { unknown-model-type }
  { Unknown~color~model~type~'#1'. }
  {
    LaTeX~has~been~asked~to~create~a~new~color~model~called~'#1',~
    but~this~type~of~model~was~never~set~up.
  }
\prop_gput:Nnn \g_msg_module_name_prop { color } { LaTeX }
```

```
\prop_gput:Nnn \g_msg_module_type_prop { color } { }
\msg_new:nnn { color } { show }
  {
    The~color~#1~
    \tl_if_empty:nTF {#2}
      { is~undefined. }
      { has~the~properties: #2 }
  }
⟨/package⟩
```

Chapter 94

l3pdf implementation

```
⟨*package⟩
⟨@@=pdf⟩
⟨*tex⟩
```

\s__pdf_stop Internal scan marks.

```
\scan_new:N \s__pdf_stop
```

(*End of definition for* \s__pdf_stop.)

\g__pdf_init_bool A boolean so we have some chance of avoiding setting things we are not allowed to. As we are potentially early in the format, we have to work a bit harder than ideal.

```
\bool_new:N \g__pdf_init_bool
\bool_lazy_and:nnT
  { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
  { \tl_if_exist_p:N \@expl@finalise@setup@@ }
  {
    \tl_gput_right:Nn \@expl@finalise@setup@@
      {
        \tl_gput_right:Nn \@kernel@after@begindocument
          { \bool_gset_true:N \g__pdf_init_bool }
      }
  }
```

(*End of definition for* \g__pdf_init_bool.)

94.1 Compression

\pdf_uncompress: Simple to do.

```
\cs_new_protected:Npn \pdf_uncompress:
  {
    \bool_if:NF \g__pdf_init_bool
      {
        \__pdf_backend_compresslevel:n { 0 }
        \__pdf_backend_compress_objects:n { \c_false_bool }
      }
  }
```

(*End of definition for* \pdf_uncompress:. *This function is documented on page* 330.)

94.2 Objects

\g__pdf_backend_object_int

For returning object numbers.

```
\int_new:N \g__pdf_backend_object_int
```

(*End of definition for* `\g__pdf_backend_object_int`.)

\pdf_object_new:n
\pdf_object_write:nnn
\pdf_object_write:nne
\pdf_object_write:nnx
\pdf_object_ref:n
__kernel_pdf_object_id:n

Simple to do: all objects create a constant `int` so it is not a backend-specific name.

```
\cs_new_protected:Npn \pdf_object_new:n #1
  {
    \__pdf_backend_object_new:
    \__pdf_object_record:nN {#1} \g__pdf_backend_object_int
  }
\cs_new_protected:Npn \pdf_object_write:nnn #1#2#3
  {
    \exp_args:Ne \__pdf_backend_object_write:nnn
      { \__pdf_object_retrieve:n {#1} } {#2} {#3}
    \bool_gset_true:N \g__pdf_init_bool
  }
\cs_generate_variant:Nn \pdf_object_write:nnn { nne , nnx }
\cs_new:Npn \pdf_object_ref:n #1
  {
    \exp_args:Ne \__pdf_backend_object_ref:n
      { \__pdf_object_retrieve:n {#1} }
  }
\cs_new:Npn \__kernel_pdf_object_id:n #1
  {
    \exp_args:Ne \__pdf_backend_object_id:n
      { \__pdf_object_retrieve:n {#1} }
  }
⟨/tex⟩
```

(*End of definition for* `\pdf_object_new:n` *and others. These functions are documented on page 328.*)

__pdf_object_record:nN
__pdf_object_retrieve:n
ltx.pdf.object_id

Object mappings are tracked in Lua for LuaTeX as this makes retrieving them much easier; as a result, there is a split in approaches. In Lua we store values in a table indexed by name. The Lua function here is set up to deal with both named and indexed objects: fits the Lua idiom well.

```
⟨*lua⟩

local scan_int = token.scan_int
local scan_string = token.scan_string
local cprint = tex.cprint

local __pdf_objects_named = {}
local __pdf_objects_indexed = {}

luacmd('__pdf_object_record:nN', function()
  local name = scan_string()
  local n = scan_int()
  __pdf_objects_named[name] = n
end,'protected','global')

local function object_id(name,index)
```

```
    if index then
      return __pdf_objects_indexed[name][index] or 0
    else
      return __pdf_objects_named[name] or 0
    end
  end

  luacmd('__pdf_object_retrieve:n', function()
    local name = scan_string()
    return cprint(12,tostring(object_id(name)))
  end,'global')

  ltx.pdf = ltx.pdf or {}
  ltx.pdf.object_id = object_id

⟨/lua⟩
```

Whereas in TeX we use integer constants.

```
⟨*tex⟩
\sys_if_engine_luatex:F
  {
    \cs_new_protected:Npn \__pdf_object_record:nN #1#2
      {
        \int_const:cn
          { c__pdf_object_ #1 _int } {#2}
      }
    \cs_new:Npn \__pdf_object_retrieve:n #1
      {
        \int_if_exist:cTF { c__pdf_object_ #1 _int }
          {
            \int_use:c
              { c__pdf_object_ #1 _int }
          }
          { 0 }
      }
  }
```

(*End of definition for* `\__pdf_object_record:nN`, `\__pdf_object_retrieve:n`, *and* `ltx.pdf.object_id`. *This function is documented on page* ??.)

`\pdf_object_if_exist_p:n`
`\pdf_object_if_exist:nTF`

```
\prg_new_conditional:Npnn \pdf_object_if_exist:n #1 { p , T , F , TF }
  {
    \int_compare:nNnTF { \__pdf_object_retrieve:n {#1} } = 0
      \prg_return_false:
      \prg_return_true:
  }
```

(*End of definition for* `\pdf_object_if_exist:nTF`. *This function is documented on page 328.*)

`\pdf_object_new_indexed:nn`
`\pdf_object_write_indexed:nnnn`
`\pdf_object_write_indexed:nnne`
`\pdf_object_ref_indexed:nn`
`\__kernel_pdf_object_id_indexed:nn`

Again we split between the common code and the macro- or Lua-based implementation. To make life easier for the Lua route, all of the potential expressions are expanded to braced numbers.

```
\cs_new_protected:Npn \pdf_object_new_indexed:nn #1#2
  {
```

```
    \__pdf_backend_object_new:
    \__pdf_object_record:neN {#1}
      { \int_eval:n {#2} } \g__pdf_backend_object_int
  }
\cs_new_protected:Npn \pdf_object_write_indexed:nnnn #1#2#3#4
  {
    \exp_args:Ne \__pdf_backend_object_write:nnn
      { \__pdf_object_retrieve:ne {#1} { \int_eval:n {#2} } } {#3} {#4}
    \bool_gset_true:N \g__pdf_init_bool
  }
\cs_generate_variant:Nn \pdf_object_write_indexed:nnnn { nnne }
\cs_new:Npn \pdf_object_ref_indexed:nn #1#2
  {
    \exp_args:Ne \__pdf_backend_object_ref:n
      { \__pdf_object_retrieve:ne {#1} { \int_eval:n {#2} } }
  }
\cs_new:Npn \__kernel_pdf_object_id_indexed:nn #1#2
  {
    \exp_args:Ne \__pdf_backend_object_id:n
      { \__pdf_object_retrieve:ne {#1} { \int_eval:n {#2} } }
  }
⟨/tex⟩
```

(*End of definition for* `\pdf_object_new_indexed:nn` *and others. These functions are documented on page* *329*.)

`\__pdf_object_record:nnN`
`\__pdf_object_record:neN`
`\__pdf_object_retrieve:nn`
`\__pdf_object_record:NnN`
`\__pdf_object_retrieve:Nn`

Again we split for Lua: the same idea as above but with nested tables. As we've arranged above that the TEX code passes a braced number, we can use `tonumber(scan_string())` rather than `scan_int()` for the index.

```
⟨*lua⟩

luacmd('__pdf_object_record:nnN', function()
  local name = scan_string()
  local index = tonumber(scan_string())
  local n = scan_int()
  __pdf_objects_indexed[name] = __pdf_objects_indexed[name] or {}
  __pdf_objects_indexed[name][index] = n
end,'protected','global')

luacmd('__pdf_object_retrieve:nn', function()
  local name = scan_string()
  local index = tonumber(scan_string())
  return cprint(12,tostring(object_id(name,index)))
end,'global')

⟨/lua⟩
```

The non-Lua approach is to divide the range into blocks, and store in integer arrays that can simulate dynamic assignment.

```
⟨*tex⟩
\sys_if_engine_luatex:F
  {
    \cs_new_protected:Npn \__pdf_object_record:nnN #1#2#3
      {
        \use:e
```

```
          {
            \__pdf_object_record:NnN
               \__pdf_object_index_split:nn {#1} {#2}
               \exp_not:N #3
          }
      }
    \cs_new_protected:Npn \__pdf_object_record:NnN #1#2#3
      {
        \intarray_if_exist:NF #1
          { \intarray_new:Nn #1 \c__pdf_object_block_size_int }
        \intarray_gset:Nnn #1 {#2} #3
      }
    \cs_new:Npn \__pdf_object_retrieve:nn #1#2
      {
        \use:e
          {
            \exp_not:N \__pdf_object_retrieve:Nn
              \__pdf_object_index_split:nn {#1} {#2}
          }
      }
    \cs_new:Npn \__pdf_object_retrieve:Nn #1#2
      { \intarray_item:Nn #1 {#2} }
```

As we want blocks to start from one, and within the block for the top value to be "in" the block, we do a little bit of manipulation. By shifting down by one, we keep the values "in" the block, then we adjust the block/index number to get back on track.

```
    \cs_new:Npn \__pdf_object_index_split:nn #1#2
      {
        \exp_not:c
          {
            g__pdf_object_ #1 _
            \int_eval:n
              {
                \int_div_truncate:nn { #2 - 1 }
                  \c__pdf_object_block_size_int + 1
              }
            _intarray
          }
          {
            \int_eval:n
              { \int_mod:nn { #2 - 1 } \c__pdf_object_block_size_int + 1 }
          }
      }
```

(*End of definition for* `\__pdf_object_record:nnN` *and others.*)

\c__pdf_object_block_size_int

Sets the block size used for managing indexed objects.

```
    \int_const:Nn \c__pdf_object_block_size_int { 10000 }
  }
```

(*End of definition for* `\c__pdf_object_block_size_int`.)

__pdf_object_record:neN
__pdf_object_retrieve:ne

Common variants.

```
\cs_generate_variant:Nn \__pdf_object_record:nnN { ne }
\cs_generate_variant:Nn \__pdf_object_retrieve:nn { ne }
```

(*End of definition for* __pdf_object_record:neN *and* __pdf_object_retrieve:ne.)

\pdf_object_unnamed_write:nn
\pdf_object_unnamed_write:ne
\pdf_object_unnamed_write:nx

No tracking needed here.

```
\cs_new_protected:Npn \pdf_object_unnamed_write:nn #1#2
  {
    \exp_args:Ne \__pdf_backend_object_now:nn {#1} {#2}
    \bool_gset_true:N \g__pdf_init_bool
  }
\cs_generate_variant:Nn \pdf_object_unnamed_write:nn { ne , nx }
```

(*End of definition for* \pdf_object_unnamed_write:nn. *This function is documented on page 329.*)

\pdf_object_ref_last:

A one-step wrapper for consistency.

```
\cs_new:Npn \pdf_object_ref_last: { \__pdf_backend_object_last: }
```

(*End of definition for* \pdf_object_ref_last:. *This function is documented on page 330.*)

\pdf_pageobject_ref:n

```
\cs_new:Npn \pdf_pageobject_ref:n #1
  { \exp_args:Ne \__pdf_backend_pageobject_ref:n {#1} }
```

(*End of definition for* \pdf_pageobject_ref:n. *This function is documented on page 330.*)

94.3 Version

\pdf_version_compare_p:Nn
\pdf_version_compare:NnTF
__pdf_version_compare_=:w
__pdf_version_compare_<:w
__pdf_version_compare_>:w

To compare version, we need to split the given value then deal with both major and minor version

```
\prg_new_conditional:Npnn \pdf_version_compare:Nn #1#2 { p , T , F , TF }
  { \use:c { __pdf_version_compare_ #1 :w } #2 . . \s__pdf_stop }
\cs_new:cpn { __pdf_version_compare_=:w } #1 . #2 . #3 \s__pdf_stop
  {
    \bool_lazy_and:nnTF
      { \int_compare_p:nNn \__pdf_backend_version_major: = {#1} }
      { \int_compare_p:nNn \__pdf_backend_version_minor: = {#2} }
      { \prg_return_true: }
      { \prg_return_false: }
  }
\cs_new:cpn { __pdf_version_compare_<:w } #1 . #2 . #3 \s__pdf_stop
  {
    \bool_lazy_or:nnTF
      { \int_compare_p:nNn \__pdf_backend_version_major: < {#1} }
      {
        \bool_lazy_and_p:nn
          { \int_compare_p:nNn \__pdf_backend_version_major: = {#1} }
          { \int_compare_p:nNn \__pdf_backend_version_minor: < {#2} }
      }
      { \prg_return_true: }
      { \prg_return_false: }
  }
\cs_new:cpn { __pdf_version_compare_>:w } #1 . #2 . #3 \s__pdf_stop
  {
    \bool_lazy_or:nnTF
      { \int_compare_p:nNn \__pdf_backend_version_major: > {#1} }
```

```
      {
        \bool_lazy_and_p:nn
          { \int_compare_p:nNn \__pdf_backend_version_major: = {#1} }
          { \int_compare_p:nNn \__pdf_backend_version_minor: > {#2} }
      }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End of definition for* `\pdf_version_compare:NnTF` *and others. This function is documented on page 330.*)

`\pdf_version_gset:n`
`\pdf_version_min_gset:n`
`\__pdf_version_gset:w`

Split the version and set.

```
\cs_new_protected:Npn \pdf_version_gset:n #1
  { \__pdf_version_gset:w #1 . . \s__pdf_stop }
\cs_new_protected:Npn \pdf_version_min_gset:n #1
  {
    \pdf_version_compare:NnT < {#1}
      { \__pdf_version_gset:w #1 . . \s__pdf_stop }
  }
\cs_new_protected:Npn \__pdf_version_gset:w #1 . #2 . #3\s__pdf_stop
  {
    \bool_if:NF \g__pdf_init_bool
      {
        \__pdf_backend_version_major_gset:n {#1}
        \__pdf_backend_version_minor_gset:n {#2}
      }
  }
```

(*End of definition for* `\pdf_version_gset:n`*,* `\pdf_version_min_gset:n`*, and* `\__pdf_version_gset:w`*. These functions are documented on page 330.*)

`\pdf_version:`
`\pdf_version_major:`
`\pdf_version_minor:`

Wrappers.

```
\cs_new:Npn \pdf_version:
  { \__pdf_backend_version_major: . \__pdf_backend_version_minor: }
\cs_new:Npn \pdf_version_major: { \__pdf_backend_version_major: }
\cs_new:Npn \pdf_version_minor: { \__pdf_backend_version_minor: }
```

(*End of definition for* `\pdf_version:`*,* `\pdf_version_major:`*, and* `\pdf_version_minor:`*. These functions are documented on page 330.*)

94.4 Page size

`\pdf_pagesize_gset:nn`

```
\cs_new_protected:Npn \pdf_pagesize_gset:nn #1#2
  { \__pdf_backend_pagesize_gset:nn {#1} {#2} }
```

(*End of definition for* `\pdf_pagesize_gset:nn`*. This function is documented on page 330.*)

94.5 Destinations

\pdf_destination:nn

```
\cs_new_protected:Npn \pdf_destination:nn #1#2
  { \__pdf_backend_destination:nn {#1} {#2} }
```

(*End of definition for* `\pdf_destination:nn`. *This function is documented on page 331.*)

\pdf_destination:nnnn

```
\cs_new_protected:Npn \pdf_destination:nnnn #1#2#3#4
  {
    \hbox_to_zero:n
      { \__pdf_backend_destination:nnnn {#1} {#2} {#3} {#4} }
  }
```

(*End of definition for* `\pdf_destination:nnnn`. *This function is documented on page 331.*)

94.6 PDF Page size (media box)

Everything here is delayed to the start of the document so that the backend will definitely be loaded.

```
\cs_if_exist:NT \@kernel@before@begindocument
  {
    \tl_gput_right:Nn \@kernel@before@begindocument
      {
        \bool_lazy_all:nT
          {
            { \cs_if_exist_p:N \stockheight }
            { \cs_if_exist_p:N \stockwidth }
            { \cs_if_exist_p:N \IfDocumentMetadataTF }
            { \IfDocumentMetadataTF { \c_true_bool } { \c_false_bool } }
            { \int_compare_p:nNn \tex_mag:D = { 1000 } }
          }
          {
            \bool_lazy_and:nnTF
              { \dim_compare_p:nNn \stockheight > { 0pt } }
              { \dim_compare_p:nNn \stockwidth  > { 0pt } }
              {
                \__pdf_backend_pagesize_gset:nn
                  \stockwidth \stockheight
              }
              {
                \bool_lazy_or:nnF
                  { \dim_compare_p:nNn \stockheight < { 0pt } }
                  { \dim_compare_p:nNn \stockwidth  < { 0pt } }
                  {
                    \bool_lazy_and:nnT
                      { \dim_compare_p:nNn \paperheight > { 0pt } }
                      { \dim_compare_p:nNn \paperwidth  > { 0pt } }
                      {
                        \__pdf_backend_pagesize_gset:nn
                          \paperwidth \paperheight
                      }
```

```
                    }
                }
            }
        }
    }
```

⟨/tex⟩

⟨/package⟩

Chapter 95

l3deprecation implementation

```
⟨*package⟩
⟨@@=deprecation⟩
```

95.1 Patching definitions to deprecate

__kernel_patch_deprecation:nnNNpn {⟨*date*⟩} {⟨*replacement*⟩} ⟨*definition*⟩ ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

defines the ⟨*function*⟩ to produce an error and run its ⟨*code*⟩.

We make \debug_on:n {deprecation} turn the ⟨*function*⟩ into an \outer error, and \debug_off:n {deprecation} restore whatever the behaviour was without \debug_on:n {deprecation}.

In the explanations below, ⟨*definition*⟩ ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩} or assignments that only differ in the scope of the ⟨*definition*⟩ will be called "the standard definition".

__kernel_patch_deprecation:nnNNpn
__deprecation_patch_aux:nnNNnn
__deprecation_warn_once:nnNnn
__deprecation_patch_aux:Nn
__deprecation_just_error:nnNN

(The parameter text is grabbed using #5#.) The arguments of __kernel_deprecation_code:nn are run upon \debug_on:n {deprecation} and \debug_off:n {deprecation}, respectively. In both scenarios we the ⟨*function*⟩ may be \outer so we undefine it with \tex_let:D before redefining it, with __kernel_deprecation_error:Nnn or with some code added shortly.

```
\cs_new_protected:Npn \__kernel_patch_deprecation:nnNNpn #1#2#3#4#5#
  { \__deprecation_patch_aux:nnNNnn {#1} {#2} #3 #4 {#5} }
\cs_new_protected:Npn \__deprecation_patch_aux:nnNNnn #1#2#3#4#5#6
  {
    \__kernel_deprecation_code:nn
      {
        \tex_let:D #4 \scan_stop:
        \__kernel_deprecation_error:Nnn #4 {#2} {#1}
      }
      { \tex_let:D #4 \scan_stop: }
    \cs_if_eq:NNTF #3 \cs_gset_protected:Npn
      { \__deprecation_warn_once:nnNnn {#1} {#2} #4 {#5} {#6} }
      { \__deprecation_patch_aux:Nn #3 { #4 #5 {#6} } }
  }
```

In case we want a warning, the ⟨*function*⟩ is defined to produce such a warning without grabbing any argument, then redefine itself to the standard definition that the ⟨*function*⟩ should have, with arguments, and call that definition. The `e`-type expansion and `\exp_not:n` avoid needing to double the `#`, which we could not do anyways. We then deal with the code for `\debug_off:n {deprecation}`: presumably someone doing that does not need the warning so we simply do the standard definition.

```
\cs_new_protected:Npn \__deprecation_warn_once:nnNnn #1#2#3#4#5
  {
    \cs_gset_protected:Npe #3
      {
        \__kernel_if_debug:TF
          {
            \exp_not:N \msg_warning:nneee
              { deprecation } { deprecated-command }
              {#1}
              { \token_to_str:N #3 }
              { \tl_to_str:n {#2} }
          }
          { }
        \exp_not:n { \cs_gset_protected:Npn #3 #4 {#5} }
        \exp_not:N #3
      }
    \__kernel_deprecation_code:nn { }
      { \cs_set_protected:Npn #3 #4 {#5} }
  }
```

In case we want neither warning nor error, the ⟨*function*⟩ is given its standard definition. Here `#1` is `\cs_new:Npn` or `\cs_new_protected:Npn`) and `#2` is ⟨*function*⟩ ⟨*parameters*⟩ `{`⟨*code*⟩`}`, so `#1#2` performs the assignment. For `\debug_off:n {deprecation}` we want to use the same assignment but with a different scope, hence the `\cs_if_eq:NNTF` test.

```
\cs_new_protected:Npn \__deprecation_patch_aux:Nn #1#2
  {
    #1 #2
    \cs_if_eq:NNTF #1 \cs_gset_protected:Npn
      { \__kernel_deprecation_code:nn { } { \cs_set_protected:Npn #2 } }
      { \__kernel_deprecation_code:nn { } { \cs_set:Npn #2 } }
  }
```

(*End of definition for* `\__kernel_patch_deprecation:nnNNpn` *and others.*)

`\__kernel_deprecation_error:Nnn`

The `\outer` definition here ensures the command cannot appear in an argument.

```
\cs_new_protected:Npn \__kernel_deprecation_error:Nnn #1#2#3
  {
    \tex_protected:D \tex_outer:D \tex_edef:D #1
      {
        \exp_not:N \msg_expandable_error:nnnnn
          { deprecation } { deprecated-command }
          { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
        \exp_not:N \msg_error:nneee
          { deprecation } { deprecated-command }
          { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
      }
  }
```

(*End of definition for* `\__kernel_deprecation_error:Nnn`.)

```
\msg_new:nnn { deprecation } { deprecated-command }
  {
    \tl_if_blank:nF {#3} { Use~ \tl_trim_spaces:n {#3} ~not~ }
    #2~deprecated~on~#1.
  }
```

95.2 Deprecated l3basics functions

```
⟨@@=cs⟩
```

`\cs_argument_spec:N`

For the present, do not deprecate fully as $\LaTeX 2_\varepsilon$ will need to catch up: one for Fall 2022.

```
%\__kernel_patch_deprecation:nnNNpn { 2022-06-24 } { \cs_parameter_spec:N }
\cs_new:Npn \cs_argument_spec:N { \cs_parameter_spec:N }
```

(*End of definition for* `\cs_argument_spec:N`.)

95.3 Deprecated l3file functions

```
⟨@@=file⟩
```

`\iow_shipout_x:Nn`
`\iow_shipout_x:Nx`
`\iow_shipout_x:cn`
`\iow_shipout_x:cx`

Previously described as `x`-type, but the hash behaviour is really `e`-type. Currently not "live" as we need to have a transition.

```
% \__kernel_patch_deprecation:nnNNpn { 2023-10-10 } { \iow_shipout_e:Nn }
\cs_new_protected:Npn \iow_shipout_x:Nn { \iow_shipout_e:Nn }
\cs_generate_variant:Nn \iow_shipout_x:Nn { Nx , c, cx }
```

(*End of definition for* `\iow_shipout_x:Nn`.)

95.4 Deprecated l3keys functions

```
⟨@@=keys⟩
```

`.str_set_x:N`
`.str_set_x:c`
`.str_gset_x:N`
`.str_gset_x:c`

```
\cs_new_protected:cpn { \c__keys_props_root_str .str_set_x:N } #1
  { \__keys_variable_set:NnnN #1 { str } { } x }
\cs_new_protected:cpn { \c__keys_props_root_str .str_set_x:c } #1
  { \__keys_variable_set:cnnN {#1} { str } { } x }
\cs_new_protected:cpn { \c__keys_props_root_str .str_gset_x:N } #1
  { \__keys_variable_set:NnnN #1 { str } { g } x }
\cs_new_protected:cpn { \c__keys_props_root_str .str_gset_x:c } #1
  { \__keys_variable_set:cnnN {#1} { str } { g } x }
```

(*End of definition for* `.str_set_x:N` *and* `.str_gset_x:N`.)

`.tl_set_x:N`
`.tl_set_x:c`
`.tl_gset_x:N`
`.tl_gset_x:c`

```
\cs_new_protected:cpn { \c__keys_props_root_str .tl_set_x:N } #1
  { \__keys_variable_set:NnnN #1 { tl } { } x }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_set_x:c } #1
  { \__keys_variable_set:cnnN {#1} { tl } { } x }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_x:N } #1
```

```
  { \__keys_variable_set:NnnN #1 { tl } { g } x }
\cs_new_protected:cpn { \c__keys_props_root_str .tl_gset_x:c } #1
  { \__keys_variable_set:cnnN {#1} { tl } { g } x }
```

(*End of definition for* `.tl_set_x:N` *and* `.tl_gset_x:N`.)

\keys_set_filter:nnnN
\keys_set_filter:nnVN
\keys_set_filter:nnvN
\keys_set_filter:nnoN
\keys_set_filter:nnnnN
\keys_set_filter:nnVnN
\keys_set_filter:nnvnN
\keys_set_filter:nnonN
\keys_set_filter:nnn
\keys_set_filter:nnV
\keys_set_filter:nnv
\keys_set_filter:nno

We need a transition here so for the present this is commented out: only needed for `latex-lab` code so this should not last for too long.

```
%\__kernel_patch_deprecation:nnNNpn { 2024-01-10 } { \keys_set_exclude_groups:nnn }
\cs_new_protected:Npn \keys_set_filter:nnn { \keys_set_exclude_groups:nnn }
\cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }
%\__kernel_patch_deprecation:nnNNpn { 2024-01-10 } { \keys_set_exclude_groups:nnnN }
\cs_new_protected:Npn  \keys_set_filter:nnnN { \keys_set_exclude_groups:nnnN }
\cs_generate_variant:Nn \keys_set_filter:nnnN { nnV , nnv , nno }
%\__kernel_patch_deprecation:nnNNpn { 2024-01-10 } { \keys_set_exclude_groups:nnnnN }
\cs_new_protected:Npn  \keys_set_filter:nnnnN { \keys_set_exclude_groups:nnnnN }
\cs_generate_variant:Nn \keys_set_filter:nnnnN { nnV , nnv , nno }
```

(*End of definition for* \keys_set_filter:nnnN *,* \keys_set_filter:nnnnN *, and* \keys_set_filter:nnn*.*)

95.5 Deprecated l3msg functions

```
⟨@@=msg⟩
```

\msg_gset:nnnn
\msg_gset:nnn

```
\__kernel_patch_deprecation:nnNNpn { 2024-02-13 } { \msg_set:nnnn }
\cs_new_protected:Npn \msg_gset:nnnn { \msg_set:nnnn }
\__kernel_patch_deprecation:nnNNpn { 2024-02-13 } { \msg_set:nnn }
\cs_new_protected:Npn \msg_gset:nnn { \msg_set:nnn }
```

(*End of definition for* \msg_gset:nnnn *and* \msg_gset:nnn*.*)

95.6 Deprecated l3pdf functions

```
⟨@@=pdf⟩
```

\g__pdf_object_prop

For tracking objects.

```
\prop_new:N \g__pdf_object_prop
```

(*End of definition for* \g__pdf_object_prop*.*)

\pdf_object_new:nn
\pdf_object_write:nn
\pdf_object_write:nx

```
\__kernel_patch_deprecation:nnNNpn { 2022-08-30 } { [\pdf_object_new:n] }
\cs_new_protected:Npn \pdf_object_new:nn #1#2
  {
    \prop_gput:Nnn \g__pdf_object_prop {#1} {#2}
    \pdf_object_new:n {#1}
  }
\__kernel_patch_deprecation:nnNNpn { 2022-08-30 } { [\pdf_object_write:n] }
\cs_new_protected:Npn \pdf_object_write:nn #1#2
  {
    \exp_args:Nee \__pdf_backend_object_write:nnn
      { \__pdf_object_retrieve:n {#1} }
      { \prop_item:Nn \g__pdf_object_prop {#1} } {#2}
```

```
    \bool_gset_true:N \g__pdf_init_bool
  }
\cs_generate_variant:Nn \pdf_object_write:nn { nx }
```

(*End of definition for* `\pdf_object_new:nn` *and* `\pdf_object_write:nn`.)

95.7 Deprecated l3prg functions

```
⟨@@=cs⟩
```

`\bool_case_true:n`
`\bool_case_true:nTF`

```
\__kernel_patch_deprecation:nnNNpn { 2023-05-03 } { \bool_case:n }
\cs_new:Npn \bool_case_true:n { \bool_case:n }
\__kernel_patch_deprecation:nnNNpn { 2023-05-03 } { \bool_case:nT }
\cs_new:Npn \bool_case_true:nT { \bool_case:nT }
\__kernel_patch_deprecation:nnNNpn { 2023-05-03 } { \bool_case:nF }
\cs_new:Npn \bool_case_true:nF { \bool_case:nF }
\__kernel_patch_deprecation:nnNNpn { 2023-05-03 } { \bool_case:nTF }
\cs_new:Npn \bool_case_true:nTF { \bool_case:nTF }
```

(*End of definition for* `\bool_case_true:nTF`.)

95.8 Deprecated l3str functions

```
⟨@@=str⟩
```

`\str_lower_case:n`
`\str_lower_case:f`
`\str_upper_case:n`
`\str_upper_case:f`
`\str_fold_case:n`
`\str_fold_case:V`

```
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_lowercase:n }
\cs_new:Npn \str_lower_case:n { \str_lowercase:n }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_lowercase:f }
\cs_new:Npn \str_lower_case:f { \str_lowercase:f }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_uppercase:n }
\cs_new:Npn \str_upper_case:n { \str_uppercase:n }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_uppercase:f }
\cs_new:Npn \str_upper_case:f { \str_uppercase:f }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_casefold:n }
\cs_new:Npn \str_fold_case:n { \str_casefold:n }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_casefold:V }
\cs_new:Npn \str_fold_case:V { \str_casefold:V }
```

(*End of definition for* `\str_lower_case:n`, `\str_upper_case:n`, *and* `\str_fold_case:n`.)

`\str_foldcase:n`
`\str_foldcase:V`

```
\__kernel_patch_deprecation:nnNNpn { 2020-10-17 } { \str_casefold:n }
\cs_new:Npn \str_foldcase:n { \str_casefold:n }
\__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \str_casefold:V }
\cs_new:Npn \str_foldcase:V { \str_casefold:V }
```

(*End of definition for* `\str_foldcase:n`.)

\str_declare_eight_bit_encoding:nnn

This command was made internal, with one more argument. There is no easy way to compute a reasonable value for that extra argument so we take a value that is big enough to accommodate all of Unicode.

```
\__kernel_patch_deprecation:nnNNpn { 2020-08-20 } { }
\cs_new_protected:Npn \str_declare_eight_bit_encoding:nnn #1
  { \__str_declare_eight_bit_encoding:nnnn {#1} { 1114112 } }
```

(*End of definition for* `\str_declare_eight_bit_encoding:nnn`*.*)

95.9 Deprecated l3seq functions

```
⟨@@=seq⟩
```

\seq_indexed_map_inline:Nn
\seq_indexed_map_function:NN

```
\__kernel_patch_deprecation:nnNNpn { 2020-06-18 } { \seq_map_indexed_inline:Nn }
\cs_new_protected:Npn \seq_indexed_map_inline:Nn { \seq_map_indexed_inline:Nn }
\__kernel_patch_deprecation:nnNNpn { 2020-06-18 } { \seq_map_indexed_function:NN }
\cs_new:Npn \seq_indexed_map_function:NN { \seq_map_indexed_function:NN }
```

(*End of definition for* `\seq_indexed_map_inline:Nn` *and* `\seq_indexed_map_function:NN`*.*)

\seq_mapthread_function:NNN

```
\__kernel_patch_deprecation:nnNNpn { 2023-05-10 } { \seq_map_pairwise_function:NNN }
\cs_new:Npn \seq_mapthread_function:NNN { \seq_map_pairwise_function:NNN }
```

(*End of definition for* `\seq_mapthread_function:NNN`*.*)

\seq_set_map_x:NNn
\seq_gset_map_x:NNn

```
\__kernel_patch_deprecation:nnNNpn { 2023-10-26 } { \seq_set_map_e:NNn }
\cs_new_protected:Npn \seq_set_map_x:NNn { \seq_set_map_e:NNn }
\__kernel_patch_deprecation:nnNNpn { 2023-10-26 } { \seq_gset_map_e:NNn }
\cs_new_protected:Npn \seq_gset_map_x:NNn { \seq_gset_map_e:NNn }
```

(*End of definition for* `\seq_set_map_x:NNn` *and* `\seq_gset_map_x:NNn`*.*)

95.10 Deprecated l3sys functions

```
⟨@@=sys⟩
```

\sys_load_deprecation:

```
\__kernel_patch_deprecation:nnNNpn { 2021-01-11 } { (no~longer~required) }
\cs_new_protected:Npn \sys_load_deprecation: { }
```

(*End of definition for* `\sys_load_deprecation:`*.*)

95.11 Deprecated l3text functions

```
⟨@@=text⟩
```

\text_titlecase:n
\text_titlecase:nn

```
\__kernel_patch_deprecation:nnNNpn { 2023-07-08 } { \text_titlecase_first:n }
\cs_new:Npn \text_titlecase:n #1
  { \text_titlecase_first:n { \text_lowercase:n {#1} } }
\__kernel_patch_deprecation:nnNNpn { 2023-07-08 } { \text_titlecase_first:nn }
\cs_new:Npn \text_titlecase:nn #1#2
  { \text_titlecase_first:nn {#1} { \text_lowercase:n {#2} } }
```

(*End of definition for* `\text_titlecase:n` *and* `\text_titlecase:nn`.)

95.12 Deprecated l3tl functions

```
⟨@@=tl⟩
```

\tl_lower_case:n
\tl_lower_case:nn
\tl_upper_case:n
\tl_upper_case:nn
\tl_mixed_case:n
\tl_mixed_case:nn

```
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_lowercase:n }
\cs_new:Npn \tl_lower_case:n #1
  { \text_lowercase:n {#1} }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_lowercase:nn }
\cs_new:Npn \tl_lower_case:nn #1#2
  { \text_lowercase:nn {#1} {#2} }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_uppercase:n }
\cs_new:Npn \tl_upper_case:n #1
  { \text_uppercase:n {#1} }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_uppercase:nn }
\cs_new:Npn \tl_upper_case:nn #1#2
  { \text_uppercase:nn {#1} {#2} }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_titlecase_first:n }
\cs_new:Npn \tl_mixed_case:n #1
  { \text_titlecase_first:n { \text_lowercase:n {#1} } }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_titlecase_first:nn }
\cs_new:Npn \tl_mixed_case:nn #1#2
  { \text_titlecase_first:nn {#1} { \text_lowercase:n {#2} } }
```

(*End of definition for* `\tl_lower_case:n` *and others.*)

\tl_case:Nn
\tl_case:cn
\tl_case:Nn*TF*
\tl_case:cn*TF*

```
\__kernel_patch_deprecation:nnNNpn { 2022-05-23 } { \token_case_meaning:Nn }
\cs_new:Npn \tl_case:Nn { \token_case_meaning:Nn }
\__kernel_patch_deprecation:nnNNpn { 2022-05-23 } { \token_case_meaning:NnT }
\cs_new:Npn \tl_case:NnT { \token_case_meaning:NnT }
\__kernel_patch_deprecation:nnNNpn { 2022-05-23 } { \token_case_meaning:NnF }
\cs_new:Npn \tl_case:NnF { \token_case_meaning:NnF }
\__kernel_patch_deprecation:nnNNpn { 2022-05-23 } { \token_case_meaning:NnTF }
\cs_new:Npn \tl_case:NnTF { \token_case_meaning:NnTF }
\cs_generate_variant:Nn \tl_case:Nn   { c }
\prg_generate_conditional_variant:Nnn \tl_case:Nn
  { c } { T , F , TF }
```

(*End of definition for* `\tl_case:NnTF`.)

\tl_build_clear:N
\tl_build_gclear:N

```
\__kernel_patch_deprecation:nnNNpn { 2023-10-18 } { \tl_build_begin:N }
\cs_new_protected:Npn \tl_build_clear:N { \tl_build_begin:N }
\__kernel_patch_deprecation:nnNNpn { 2023-10-18 } { \tl_build_gbegin:N }
\cs_new_protected:Npn \tl_build_gclear:N { \tl_build_gbegin:N }
```

(*End of definition for* `\tl_build_clear:N` *and* `\tl_build_gclear:N`*.*)

\tl_build_get:NN

```
\__kernel_patch_deprecation:nnNNpn { 2023-10-25 } { \tl_build_get_intermediate:NN }
\cs_new_protected:Npn \tl_build_get:NN { \tl_build_get_intermediate:NN }
```

(*End of definition for* `\tl_build_get:NN`*.*)

95.13 Deprecated l3token functions

```
⟨@@=char⟩
```

\char_to_utfviii_bytes:n

```
\__kernel_patch_deprecation:nnNNpn { 2022-10-09 } { [ \codepoint_generate:nn ] }
\cs_new:Npn \char_to_utfviii_bytes:n { \__kernel_codepoint_to_bytes:n }
```

(*End of definition for* `\char_to_utfviii_bytes:n`*.*)

\char_to_nfd:N
\char_to_nfd:n

```
\__kernel_patch_deprecation:nnNNpn { 2022-10-09 } { \codepoint_to_nfd:n }
\cs_new:Npn \char_to_nfd:N #1 { \codepoint_to_nfd:n {`#1} }
\__kernel_patch_deprecation:nnNNpn { 2022-10-09 } { \codepoint_to_nfd:n }
\cs_new:Npn \char_to_nfd:n { \codepoint_to_nfd:n }
```

(*End of definition for* `\char_to_nfd:N` *and* `\char_to_nfd:n`*.*)

\char_lower_case:N
\char_upper_case:N
\char_mixed_case:Nn
\char_fold_case:N
\char_str_lower_case:N
\char_str_upper_case:N
\char_str_mixed_case:N
\char_str_fold_case:N

```
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_lowercase:n }
\cs_new:Npn \char_lower_case:N { \text_lowercase:n }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_uppercase:n }
\cs_new:Npn \char_upper_case:N { \text_uppercase:n }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_titlecase_first:n }
\cs_new:Npn \char_mixed_case:N { \text_titlecase_first:n }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_casefold:n }
\cs_new:Npn \char_fold_case:N { \str_casefold:n }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_lowercase:n }
\cs_new:Npn \char_str_lower_case:N { \str_lowercase:n }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_uppercase:n }
\cs_new:Npn \char_str_upper_case:N { \str_uppercase:n }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_titlecase:n }
\cs_new:Npn \char_str_mixed_case:N { \str_titlecase:n }
\__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_casefold:n }
\cs_new:Npn \char_str_fold_case:N { \str_casefold:n }
```

(*End of definition for* `\char_lower_case:N` *and others.*)

\char_lowercase:N
\char_titlecase:N
\char_uppercase:N
\char_foldcase:N
\char_str_lowercase:N
\char_str_titlecase:N
\char_str_uppercase:N
\char_str_foldcase:N

```
\__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \text_lowercase:n }
\cs_new:Npn \char_lowercase:N { \text_lowercase:n }
\__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \text_uppercase:n }
\cs_new:Npn \char_uppercase:N { \text_uppercase:n }
\__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \text_titlecase_first:n }
\cs_new:Npn \char_titlecase:N { \text_titlecase_first:n }
\__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \str_casefold:n }
\cs_new:Npn \char_foldcase:N { \str_casefold:n }
\__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \str_lowercase:n }
\cs_new:Npn \char_str_lowercase:N { \str_lowercase:n }
\__kernel_patch_deprecation:nnNNpn { 2022-10-17 }
  { \tl_to_str:e { \text_titlecase_first:n } }
\cs_new:Npn \char_str_titlecase:N #1
  { \tl_to_str:e { \text_titlecase_first:n {#1} } }
\__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \str_uppercase:n }
\cs_new:Npn \char_str_uppercase:N { \str_uppercase:n }
\__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \str_casefold:n }
\cs_new:Npn \char_str_foldcase:N { \str_casefold:n }
```

(*End of definition for* `\char_lowercase:N` *and others.*)

\peek_catcode_ignore_spaces:N*TF*
\peek_catcode_remove_ignore_spaces:N*TF*
\peek_charcode_ignore_spaces:N*TF*
\peek_charcode_remove_ignore_spaces:N*TF*
\peek_meaning_ignore_spaces:N*TF*
\peek_meaning_remove_ignore_spaces:N*TF*

A little extra fun here to deal with the expansion.

```
\tl_map_inline:nn
  {
    { catcode } { catcode_remove }
    { charcode } { charcode_remove }
    { meaning } { meaning_remove }
  }
  {
    \use:e
      {
        \__kernel_patch_deprecation:nnNNpn { 2022-01-11 } { \peek_remove_spaces:n }
        \cs_gset_protected:Npn \exp_not:c { peek_ #1 _ignore_spaces:NTF } ##1##2##3
          {
            \peek_remove_spaces:n
              { \exp_not:c { peek_ #1 :NTF } ##1 {##2} {##3} }
          }
        \__kernel_patch_deprecation:nnNNpn { 2022-01-11 } { \peek_remove_spaces:n }
        \cs_gset_protected:Npn \exp_not:c { peek_ #1 _ignore_spaces:NT } ##1##2
          {
            \peek_remove_spaces:n
              { \exp_not:c { peek_ #1 :NT } ##1 {##2} }
          }
        \__kernel_patch_deprecation:nnNNpn { 2022-01-11 } { \peek_remove_spaces:n }
        \cs_gset_protected:Npn \exp_not:c { peek_ #1 _ignore_spaces:NF } ##1##2
          {
            \peek_remove_spaces:n
              { \exp_not:c { peek_ #1 :NF } ##1 {##2} }
          }
      }
  }
```

(*End of definition for* `\peek_catcode_ignore_spaces:NTF` *and others.*)

95.14 Deprecated l3prop functions

\prop_put_if_new:Nnn
\prop_put_if_new:NVn
\prop_put_if_new:NnV
\prop_put_if_new:cnn
\prop_put_if_new:cVn
\prop_put_if_new:cnV
\prop_gput_if_new:Nnn
\prop_gput_if_new:NVn
\prop_gput_if_new:NnV
\prop_gput_if_new:cnn
\prop_gput_if_new:cVn
\prop_gput_if_new:cnV

```
%\__kernel_patch_deprecation:nnNNpn { 2024-03-30 } { \prop_put_if_not_in:Nnn }
\cs_new_protected:Npn \prop_put_if_new:Nnn { \prop_put_if_not_in:Nnn }
%\__kernel_patch_deprecation:nnNNpn { 2024-03-30 } { \prop_gput_if_not_in:Nnn }
\cs_new_protected:Npn \prop_gput_if_new:Nnn { \prop_gput_if_not_in:Nnn }
\cs_generate_variant:Nn \prop_put_if_new:Nnn
  { NnV , NV , c , cnV , cV }
\cs_generate_variant:Nn \prop_gput_if_new:Nnn
  { NnV , NV , c , cnV , cV }
```

(*End of definition for* `\prop_put_if_new:Nnn` *and* `\prop_gput_if_new:Nnn`*.*)

```
⟨/package⟩
```

Chapter 96

l3debug implementation

Internal kernel functions that are only defined here are listed in l3kernel-functions, see 41.1.

```
⟨*package⟩
⟨@@=debug⟩
```

Standard file identification.

```
\ProvidesExplFile{l3debug.def}{2024-11-02}{}{L3 Debugging support}
```

\s__debug_stop

Internal scan marks.

```
\scan_new:N \s__debug_stop
```

(*End of definition for* \s__debug_stop.)

__debug_use_i_delimit_by_s_stop:nw

Functions to gobble up to a scan mark.

```
\cs_new:Npn \__debug_use_i_delimit_by_s_stop:nw #1 #2 \s__debug_stop {#1}
```

(*End of definition for* __debug_use_i_delimit_by_s_stop:nw.)

\q__debug_recursion_tail
\q__debug_recursion_stop

Internal quarks.

```
\quark_new:N \q__debug_recursion_tail
\quark_new:N \q__debug_recursion_stop
```

(*End of definition for* \q__debug_recursion_tail *and* \q__debug_recursion_stop.)

__debug_if_recursion_tail_stop:N

Functions to query recursion quarks.

```
\cs_new:Npn \__debug_use_none_delimit_by_q_recursion_stop:w
  #1 \q__debug_recursion_stop { }
\__kernel_quark_new_test:N \__debug_if_recursion_tail_stop:N
```

(*End of definition for* __debug_if_recursion_tail_stop:N.)

\debug_on:n
\debug_off:n
__debug_all_on:
__debug_all_off:

```
\cs_gset_protected:Npn \debug_on:n #1
  {
    \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
      {
        \cs_if_exist_use:cF { __debug_ ##1 _on: }
          { \msg_error:nnn { debug } { debug } {##1} }
      }
```

```
  }
\cs_gset_protected:Npn \debug_off:n #1
  {
    \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
      {
        \cs_if_exist_use:cF { __debug_ ##1 _off: }
          { \msg_error:nnn { debug } { debug } {##1} }
      }
  }
\cs_new_protected:Npn \__debug_all_on:
  {
    \debug_on:n
      {
        check-declarations ,
        check-expressions ,
        deprecation ,
        log-functions ,
      }
  }
\cs_new_protected:Npn \__debug_all_off:
  {
    \debug_off:n
      {
        check-declarations ,
        check-expressions ,
        deprecation ,
        log-functions ,
      }
  }
```

(*End of definition for* `\debug_on:n` *and others. These functions are documented on page 31.*)

`\debug_suspend:`
`\debug_resume:`
`\__debug_suspended:T`
`\l__debug_suspended_tl`

Suspend and resume locally all debug-related errors and logging except deprecation errors. The `\debug_suspend:` and `\debug_resume:` pairs can be nested. We keep track of nesting in a token list containing a number of periods. At first begin with the "non-suspended" version of `\__debug_suspended:T`.

```
\tl_new:N \l__debug_suspended_tl { }
\cs_gset_protected:Npn \debug_suspend:
  {
    \tl_put_right:Nn \l__debug_suspended_tl { . }
    \cs_set_eq:NN \__debug_suspended:T \use:n
  }
\cs_gset_protected:Npn \debug_resume:
  {
    \__kernel_tl_set:Nx \l__debug_suspended_tl
      { \tl_tail:N \l__debug_suspended_tl }
    \tl_if_empty:NT \l__debug_suspended_tl
      {
        \cs_set_eq:NN \__debug_suspended:T \use_none:n
      }
  }
\cs_new_eq:NN \__debug_suspended:T \use_none:n
```

(*End of definition for* `\debug_suspend:` *and others. These functions are documented on page 31.*)

__debug_check-declarations_on:
__debug_check-declarations_off:
__kernel_chk_var_exist:N
__kernel_chk_cs_exist:N
__kernel_chk_cs_exist:c
__kernel_chk_flag_exist:NN
__kernel_chk_var_local:N
__kernel_chk_var_global:N
__kernel_chk_var_scope:NN

When debugging is enabled these two functions set up functions that test their argument (when `check-declarations` is active)

- `\__kernel_chk_var_exist:N` and `\__kernel_chk_cs_exist:N`, two functions that test that their argument is defined;
- `\__kernel_chk_var_scope:NN` that checks that its argument `#2` has scope `#1`.
- `\__kernel_chk_var_local:N` and `\__kernel_chk_var_global:N` that perform both checks.

```
\cs_new_protected:Npn \__kernel_chk_var_exist:N #1 { }
\cs_new_protected:Npn \__kernel_chk_cs_exist:N #1 { }
\cs_generate_variant:Nn \__kernel_chk_cs_exist:N { c }
\cs_new:Npn \__kernel_chk_flag_exist:NN { }
\cs_new_protected:Npn \__kernel_chk_var_local:N #1 { }
\cs_new_protected:Npn \__kernel_chk_var_global:N #1 { }
\cs_new_protected:Npn \__kernel_chk_var_scope:NN #1#2 { }
\cs_new_protected:cpn { __debug_check-declarations_on: }
  {
    \cs_set_protected:Npn \__kernel_chk_var_exist:N ##1
      {
        \__debug_suspended:T \use_none:nnn
        \cs_if_exist:NF ##1
          {
            \msg_error:nne { debug } { non-declared-variable }
              { \token_to_str:N ##1 }
          }
      }
    \cs_set_protected:Npn \__kernel_chk_cs_exist:N ##1
      {
        \__debug_suspended:T \use_none:nnn
        \cs_if_exist:NF ##1
          {
            \msg_error:nne { kernel } { command-not-defined }
              { \token_to_str:N ##1 }
          }
      }
    \cs_set:Npn \__kernel_chk_flag_exist:NN ##1##2
      {
        \__debug_suspended:T \use_iii:nnnn
        \flag_if_exist:NTF ##2
          { ##1 ##2 }
          {
            \msg_expandable_error:nnn { kernel } { bad-variable } {##2}
            ##1 \l_tmpa_flag
          }
      }
    \cs_set_protected:Npn \__kernel_chk_var_scope:NN
      {
        \__debug_suspended:T \use_none:nnn
        \__debug_chk_var_scope_aux:NN
      }
    \cs_set_protected:Npn \__kernel_chk_var_local:N ##1
      {
```

```
      \__debug_suspended:T \use_none:nnnnn
      \__kernel_chk_var_exist:N ##1
      \__debug_chk_var_scope_aux:NN l ##1
    }
  \cs_set_protected:Npn \__kernel_chk_var_global:N ##1
    {
      \__debug_suspended:T \use_none:nnnnn
      \__kernel_chk_var_exist:N ##1
      \__debug_chk_var_scope_aux:NN g ##1
    }
  }
\cs_new_protected:cpn { __debug_check-declarations_off: }
  {
    \cs_set_protected:Npn \__kernel_chk_var_exist:N ##1 { }
    \cs_set_protected:Npn \__kernel_chk_cs_exist:N ##1 { }
    \cs_set:Npn \__kernel_chk_flag_exist:NN { }
    \cs_set_protected:Npn \__kernel_chk_var_local:N ##1 { }
    \cs_set_protected:Npn \__kernel_chk_var_global:N ##1 { }
    \cs_set_protected:Npn \__kernel_chk_var_scope:NN ##1##2 { }
  }
```

(*End of definition for* `\__debug_check-declarations_on:` *and others.*)

__debug_chk_var_scope_aux:NN
__debug_chk_var_scope_aux:Nn
__debug_chk_var_scope_aux:NNn

First check whether the name of the variable `#2` starts with ⟨*letter*⟩`_`. If it does then pass that letter, the ⟨*scope*⟩, and the variable name to `\__debug_chk_var_scope_aux:NNn`. That function compares the two letters and triggers an error if they differ (the `\scan_stop:` case is not reachable here). If the second character was not `_` then pass the same data to the same auxiliary, except for its first argument which is now a control sequence. That control sequence is actually a token list (but to avoid triggering the checking code we manipulate it using `\cs_set_nopar:Npn`) containing a single letter ⟨*scope*⟩ according to what the first assignment to the given variable was.

```
\cs_new_protected:Npn \__debug_chk_var_scope_aux:NN #1#2
  { \exp_args:NNf \__debug_chk_var_scope_aux:Nn #1 { \cs_to_str:N #2 } }
\cs_new_protected:Npn \__debug_chk_var_scope_aux:Nn #1#2
  {
    \if:w _ \use_i:nn \__debug_use_i_delimit_by_s_stop:nw #2 ? ? \s__debug_stop
      \exp_after:wN \__debug_chk_var_scope_aux:NNn
        \__debug_use_i_delimit_by_s_stop:nw #2 ? \s__debug_stop
        #1 {#2}
    \else:
      \exp_args:Nc \__debug_chk_var_scope_aux:NNn
        { __debug_chk_/ #2 }
        #1 {#2}
    \fi:
  }
\cs_new_protected:Npn \__debug_chk_var_scope_aux:NNn #1#2#3
  {
    \if:w #1 #2
    \else:
      \if:w #1 \scan_stop:
        \cs_gset_nopar:Npn #1 {#2}
      \else:
        \msg_error:nneee { debug } { local-global }
          {#1} {#2} { \iow_char:N \\ #3 }
```

```
      \fi:
    \fi:
  }
\use:c { __debug_check-declarations_off: }
```

(*End of definition for* `\__debug_chk_var_scope_aux:NN`*,* `\__debug_chk_var_scope_aux:Nn`*, and* `\__debug_chk_var_scope_aux:NNn`*.*)

`\__debug_log-functions_on:`
`\__debug_log-functions_off:`
`\__kernel_debug_log:e`

These two functions (corresponding to the expl3 option `log-functions`) control whether `\__kernel_debug_log:e` writes to the log file or not. By default, logging is off.

```
\cs_new_protected:cpn { __debug_log-functions_on: }
  {
    \cs_set_protected:Npn \__kernel_debug_log:e
      { \__debug_suspended:T \use_none:nn \iow_log:e }
  }
\cs_new_protected:cpn { __debug_log-functions_off: }
  { \cs_set_protected:Npn \__kernel_debug_log:e { \use_none:n } }
\cs_new_protected:Npn \__kernel_debug_log:e { \use_none:n }
```

(*End of definition for* `\__debug_log-functions_on:`*,* `\__debug_log-functions_off:`*, and* `\__kernel_debug_log:e`*.*)

`\__debug_check-expressions_on:`
`\__debug_check-expressions_off:`
`\__kernel_chk_expr:nNnN`
`\__debug_chk_expr_aux:nNnN`

When debugging is enabled these two functions set `\__kernel_chk_expr:nNnN` to test or not whether the given expression is valid. The idea is to evaluate the expression within a brace group (to catch trailing `\use_none:nn` or similar), then test that the result is what we expect. This is done by turning it to an integer and hitting that with `\tex_romannumeral:D` after replacing the first character by `-0`. If all goes well, that primitive finds a non-positive integer and gives an empty output. If the original expression evaluation stopped early it leaves a trailing `\tex_relax:D`, which stops the second evaluation (used to convert to integer) before it encounters the final `\tex_relax:D`. Since `\tex_romannumeral:D` does not absorb `\tex_relax:D` the output will be nonempty. Note that `#3` is empty except for mu expressions for which it is `\tex_mutoglue:D` to avoid an "incompatible glue units" error. Note also that if we had omitted the first `\tex_relax:D` then for instance `1+2\relax+3` would incorrectly be accepted as a valid integer expression.

```
\cs_new_protected:cpn { __debug_check-expressions_on: }
  {
    \cs_set:Npn \__kernel_chk_expr:nNnN ##1##2
      {
        \__debug_suspended:T { ##1 \use_none:nnnnnnn }
        \exp_after:wN \__debug_chk_expr_aux:nNnN
        \exp_after:wN { \tex_the:D ##2 ##1 \scan_stop: }
        ##2
      }
  }
\cs_new_protected:cpn { __debug_check-expressions_off: }
  { \cs_set:Npn \__kernel_chk_expr:nNnN ##1##2##3##4 {##1} }
\cs_new:Npn \__kernel_chk_expr:nNnN #1#2#3#4 {#1}
\cs_new:Npn \__debug_chk_expr_aux:nNnN #1#2#3#4
  {
    \tl_if_empty:oF
      {
        \tex_romannumeral:D - 0
        \exp_after:wN \use_none:n
```

```
        \int_value:w #3 #2 #1 \scan_stop:
      }
      {
        \msg_expandable_error:nnnn
          { debug } { expr } {#4} {#1}
      }
    #1
  }
```

(*End of definition for* `\__debug_check-expressions_on:` *and others.*)

`\__debug_deprecation_on:`
`\__debug_deprecation_off:`

Make deprecated commands throw errors if the user requests it. This relies on two token lists, filled up in l3deprecation by calls to `\__kernel_deprecation_code:nn`.

```
\cs_new_protected:Npn \__debug_deprecation_on:
  { \g__debug_deprecation_on_tl }
\cs_new_protected:Npn \__debug_deprecation_off:
  { \g__debug_deprecation_off_tl }
```

(*End of definition for* `\__debug_deprecation_on:` *and* `\__debug_deprecation_off:`.)

`\l__debug_internal_tl`
`\l__debug_tmpa_tl`
`\l__debug_tmpb_tl`

For patching.

```
\tl_new:N \l__debug_internal_tl
\tl_new:N \l__debug_tmpa_tl
\tl_new:N \l__debug_tmpb_tl
```

(*End of definition for* `\l__debug_internal_tl` *,* `\l__debug_tmpa_tl` *, and* `\l__debug_tmpb_tl`.)

`\__debug_generate_parameter_list:NNN`
`\__debug_build_parm_text:n`
`\__debug_build_arg_list:n`
`\__debug_arg_list_from_signature:nNN`
`\__debug_arg_check_invalid:N`
`\__debug_parm_terminate:w`
`\__debug_arg_if_braced:n`
`\__debug_get_base_form:N`
`\__debug_arg_return:N`
`\__debug_arg_if_braced:NTF`

Some functions don't take the arguments their signature indicates. For instance, `\clist_concat:NNN` doesn't take (directly) any argument, so patching it with something that uses `#1`, `#2`, or `#3` results in "Illegal parameter number in definition of `\clist_-concat:NNN`".

Instead of changing *the* definition of the macros, we'll create a copy of such macros, say, `\__debug_clist_concat:NNN` which will be defined as `<debug code with #1, #2 and #3>\clist_` For that we need to identify the signature of every function and build the appropriate parameter list.

`\__debug_generate_parameter_list:NNN` takes a function in `#1` and returns teo parameter lists: `#2` contains the simple `#1#2#3` as would be used in the ⟨*parameter text*⟩ of the definition and `#3` contains the same parameters but with braces where necessary.

With the current implementation the resulting `#3` is, for example for `\some_function:NnNn`, `#1{#2}#3{#4}`. While this is correct, it might be unnecessary. Bracing everything will usually have the same outcome (unless the function was misused in the first place). What should be done?

```
\cs_new_protected:Npn \__debug_generate_parameter_list:NNN #1#2#3
  {
    \__kernel_tl_set:Nx \l__debug_internal_tl
      { \exp_last_unbraced:Nf \use_ii:nnn \cs_split_function:N #1 }
    \__kernel_tl_set:Nx #2
      { \exp_args:NV \__debug_build_parm_text:n \l__debug_internal_tl }
    \__kernel_tl_set:Nx #3
      { \exp_args:NV \__debug_build_arg_list:n \l__debug_internal_tl }
  }
\cs_new:Npn \__debug_build_parm_text:n #1
  {
```

```
    \__debug_arg_list_from_signature:nNN { 1 } \c_false_bool #1
    \q__debug_recursion_tail \q__debug_recursion_stop
  }
\cs_new:Npn \__debug_build_arg_list:n #1
  {
    \__debug_arg_list_from_signature:nNN { 1 } \c_true_bool #1
    \q__debug_recursion_tail \q__debug_recursion_stop
  }
\cs_new:Npn \__debug_arg_list_from_signature:nNN #1 #2 #3
  {
    \__debug_if_recursion_tail_stop:N #3
    \__debug_arg_check_invalid:N #3
    \bool_if:NT #2 { \__debug_arg_if_braced:NT #3 { \use_none:n } }
    \use:n { \c_hash_str \int_eval:n {#1} }
    \exp_args:Nf \__debug_arg_list_from_signature:nNN
      { \int_eval:n {#1+1} } #2
  }
```

Argument types `w`, `p`, `T`, and `F` shouldn't be included in the parameter lists, so we abort the loop if either is found.

```
\cs_new:Npn \__debug_arg_check_invalid:N #1
  {
    \if:w w #1 \__debug_parm_terminate:w \else:
      \if:w p #1 \__debug_parm_terminate:w \else:
        \if:w T #1 \__debug_parm_terminate:w \else:
          \if:w F #1 \__debug_parm_terminate:w \else:
            \exp:w
          \fi:
        \fi:
      \fi:
    \fi:
    \exp_end:
  }
\cs_new:Npn \__debug_parm_terminate:w
  { \exp_after:wN \__debug_use_none_delimit_by_q_recursion_stop:w \exp:w }
\prg_new_conditional:Npnn \__debug_arg_if_braced:N #1 { T }
  { \exp_args:Nf \__debug_arg_if_braced:n { \__debug_get_base_form:N #1 } }
\cs_new:Npn \__debug_arg_if_braced:n #1
  {
    \if:w n #1 \prg_return_true: \else:
      \if:w N #1 \prg_return_false: \else:
        \msg_expandable_error:nnn
          { debug } { bad-arg-type } {#1}
      \fi:
    \fi:
  }
\msg_new:nnn { debug } { bad-arg-type }
  { Wrong~argument~type~#1. }
```

The macro below gets the base form of an argument type given a variant. It serves only to differentiate arguments which should be braced from ones which shouldn't. If all were to be braced this would be unnecessary. I moved the `n` and `N` variants to the beginning of the test as the are much more common here.

```
\cs_new:Npn \__debug_get_base_form:N #1
  {
```

```
    \if:w n #1 \__debug_arg_return:N n \else:
      \if:w N #1 \__debug_arg_return:N N \else:
        \if:w c #1 \__debug_arg_return:N N \else:
          \if:w o #1 \__debug_arg_return:N n \else:
            \if:w V #1 \__debug_arg_return:N n \else:
              \if:w v #1 \__debug_arg_return:N n \else:
                \if:w f #1 \__debug_arg_return:N n \else:
                  \if:w e #1 \__debug_arg_return:N n \else:
                    \if:w x #1 \__debug_arg_return:N n \else:
                      \__debug_arg_return:N \scan_stop:
                    \fi:
                  \fi:
                \fi:
              \fi:
            \fi:
          \fi:
        \fi:
      \fi:
    \fi:
    \exp_stop_f:
  }
\cs_new:Npn \__debug_arg_return:N #1
  { \exp_after:wN #1 \exp:w \exp_end_continue_f:w }
```

(*End of definition for* `\__debug_generate_parameter_list:NNN` *and others.*)

`\__kernel_patch:nnn`
`\__kernel_patch_aux:nnn`
`\__debug_setup_debug_code:Nnn`
`\__debug_add_to_debug_code:Nnn`
`\__debug_insert_debug_code:Nnn`
`\__kernel_patch_weird:nnn`
`\__kernel_patch_weird_aux:nnn`
`\__debug_patch_weird:Nnn`

Simple patching by adding material at the start and end of (a collection of) functions is straight-forward as we know the catcode set up. The approach is essentially that in etoolbox. Notice the need to worry about spaces: those are otherwise lost as normally in expl3 code they would be `~`.

As discussed above, some functions don't take arguments, so we can't patch something that uses an argument in them. For these functions `\__kernel_patch:nnn` is used. It starts by creating a copy of the function (say, `\clist_concat:NNN`) with a `__debug_` prefix in the name. This copy won't be changed. The code redefines the original function to take the exact same arguments as advertised in its signature (see `\__debug_generate_parameter_list:NNN` above). The redefined function also contains the debug code in the proper position. If a function with the same name and the `__debug_` prefix was already defined, then the macro patches that definition by adding more debug code to it.

```
\group_begin:
  \cs_set_protected:Npn \__kernel_patch:nnn
    {
      \group_begin:
        \char_set_catcode_other:N \#
        \__kernel_patch_aux:nnn
    }
  \cs_set_protected:Npn \__kernel_patch_aux:nnn #1#2#3
    {
        \char_set_catcode_parameter:N \#
        \char_set_catcode_space:N \ %
        \tex_endlinechar:D -1 \scan_stop:
        \tl_map_inline:nn {#3}
          {
```

```
          \cs_if_exist:cTF { __debug_ \cs_to_str:N ##1 }
            { \__debug_add_to_debug_code:Nnn }
            { \__debug_setup_debug_code:Nnn }
              ##1 {#1} {#2}
        }
      \group_end:
    }
  \cs_set_protected:Npn \__debug_setup_debug_code:Nnn #1#2#3
    {
      \cs_gset_eq:cN { __debug_ \cs_to_str:N #1 } #1
      \__debug_generate_parameter_list:NNN #1 \l__debug_tmpa_tl \l__debug_tmpb_tl
      \exp_args:Ne \tex_scantokens:D
        {
          \tex_global:D \cs_prefix_spec:N #1
          \tex_def:D \exp_not:N #1
          \tl_use:N \l__debug_tmpa_tl
            {
              \tl_to_str:n {#2}
              \exp_not:c { __debug_ \cs_to_str:N #1 }
              \tl_use:N \l__debug_tmpb_tl
              \tl_to_str:n {#3}
            }
        }
    }
  \cs_set_protected:Npn \__debug_add_to_debug_code:Nnn #1#2#3
    {
      \use:e
        {
          \cs_set:Npn \exp_not:N \__debug_tmp:w
            ##1 \tl_to_str:n { macro: }
            ##2 \tl_to_str:n { -> }
            ##3 \c_backslash_str \tl_to_str:n { __debug_ }
                    \cs_to_str:N #1
            ##4 \s__debug_stop
            {
              \exp_not:N \exp_args:Ne \exp_not:N \tex_scantokens:D
                {
                  \tex_global:D ##1
                  \tex_def:D \exp_not:N #1 ##2
                    {
                      ##3 \tl_to_str:n {#2}
                      \c_backslash_str __debug_ \cs_to_str:N #1
                      ##4 \tl_to_str:n {#3}
                    }
                }
            }
        }
      \exp_after:wN \__debug_tmp:w \cs_meaning:N #1 \s__debug_stop
    }
```

Some functions, however, won't work with the signature reading setup above because their signature contains weird arguments. These functions need to be patched using `\__kernel_patch_weird:nnn`, which won't make a copy of the function, rather it will patch the debug code directly into it. This means that whatever argument the debug code uses must be actually used by the patched function.

```
\cs_set_protected:Npn \__kernel_patch_weird:nnn
  {
    \group_begin:
      \char_set_catcode_other:N \#
      \__kernel_patch_weird_aux:nnn
  }
\cs_set_protected:Npn \__kernel_patch_weird_aux:nnn #1#2#3
  {
      \char_set_catcode_parameter:N \#
      \char_set_catcode_space:N \ %
      \tex_endlinechar:D -1 \scan_stop:
      \tl_map_inline:nn {#3}
        { \__debug_patch_weird:Nnn ##1 {#1} {#2} }
    \group_end:
  }
\cs_set_protected:Npn \__debug_patch_weird:Nnn #1#2#3
  {
    \use:e
      {
        \tex_endlinechar:D -1 \scan_stop:
        \exp_not:N \tex_scantokens:D
          {
            \tex_global:D \cs_prefix_spec:N #1
            \tex_def:D \exp_not:N #1
            \cs_parameter_spec:N #1
              {
                \tl_to_str:n {#2}
                \cs_replacement_spec:N #1
                \tl_to_str:n {#3}
              }
          }
      }
  }
```

(*End of definition for* `\__kernel_patch:nnn` *and others.*)

Patching the second argument to ensure it exists. This happens before we alter `#1` so the ordering is correct. For many variable types such as `int` a low-level error occurs when `#2` is unknown, so adding a check is not needed.

```
\__kernel_patch:nnn
  { \__kernel_chk_var_exist:N #2 }
  { }
  {
    \bool_set_eq:NN
    \bool_gset_eq:NN
    \clist_set_eq:NN
    \clist_gset_eq:NN
    \fp_set_eq:NN
    \fp_gset_eq:NN
    \prop_set_eq:NN
    \prop_gset_eq:NN
    \seq_set_eq:NN
    \seq_gset_eq:NN
    \str_set_eq:NN
```

```
      \str_gset_eq:NN
      \tl_set_eq:NN
      \tl_gset_eq:NN
    }
```

Patching both second and third arguments.

```
  \__kernel_patch:nnn
    {
      \__kernel_chk_var_exist:N #2
      \__kernel_chk_var_exist:N #3
    }
    { }
    {
      \clist_concat:NNN
      \clist_gconcat:NNN
      \prop_concat:NNN
      \prop_gconcat:NNN
      \seq_concat:NNN
      \seq_gconcat:NNN
      \str_concat:NNN
      \str_gconcat:NNN
      \tl_concat:NNN
      \tl_gconcat:NNN
    }
```

```
\cs_gset_protected:Npn \__kernel_tl_set:Nx  { \cs_set_nopar:Npe }
\cs_gset_protected:Npn \__kernel_tl_gset:Nx { \cs_gset_nopar:Npe }
```

Patching where the first argument to a function needs scope-checking: either local or global (so two lists).

```
  \__kernel_patch:nnn
    { \__kernel_chk_var_local:N #1 }
    { }
    {
      \bool_set:Nn
      \bool_set_eq:NN
      \bool_set_true:N
      \bool_set_false:N
      \box_set_eq:NN
      \box_set_eq_drop:NN
      \box_set_to_last:N
      \clist_clear:N
      \clist_set_eq:NN
      \dim_zero:N
      \dim_set:Nn
      \dim_set_eq:NN
      \dim_add:Nn
      \dim_sub:Nn
      \fp_set_eq:NN
      \int_zero:N
      \int_set_eq:NN
      \int_add:Nn
      \int_sub:Nn
      \int_incr:N
      \int_decr:N
```

```
    \int_set:Nn
    \hbox_set:Nn
    \hbox_set_to_wd:Nnn
    \hbox_set:Nw
    \hbox_set_to_wd:Nnw
    \muskip_zero:N
    \muskip_set:Nn
    \muskip_add:Nn
    \muskip_sub:Nn
    \muskip_set_eq:NN
    \prop_clear:N
    \prop_concat:NNN
    \prop_pop:NnN
    \prop_pop:NnNT
    \prop_pop:NnNF
    \prop_pop:NnNTF
    \prop_put:Nnn
    \prop_put_if_not_in:Nnn
    \prop_put_from_keyval:Nn
    \prop_remove:Nn
    \prop_set_eq:NN
    \prop_set_from_keyval:Nn
    \seq_set_eq:NN
    \skip_zero:N
    \skip_set:Nn
    \skip_set_eq:NN
    \skip_add:Nn
    \skip_sub:Nn
    \str_clear:N
    \str_set_eq:NN
    \str_put_left:Nn
    \str_put_right:Nn
    \__kernel_tl_set:Nx
    \tl_clear:N
    \tl_set_eq:NN
    \tl_put_left:Nn
    \tl_put_left:NV
    \tl_put_left:Nv
    \tl_put_left:Ne
    \tl_put_left:No
    \tl_put_right:Nn
    \tl_put_right:NV
    \tl_put_right:Nv
    \tl_put_right:Ne
    \tl_put_right:No
    \tl_build_begin:N
    \tl_build_put_right:Nn
    \tl_build_put_left:Nn
    \vbox_set:Nn
    \vbox_set_top:Nn
    \vbox_set_to_ht:Nnn
    \vbox_set:Nw
    \vbox_set_to_ht:Nnw
    \vbox_set_split_to_ht:NNn
```

```
  }
\__kernel_patch:nnn
  { \__kernel_chk_var_global:N #1 }
  { }
  {
    \bool_gset:Nn
    \bool_gset_eq:NN
    \bool_gset_true:N
    \bool_gset_false:N
    \box_gset_eq:NN
    \box_gset_eq_drop:NN
    \box_gset_to_last:N
    \cctab_gset:Nn
    \clist_gclear:N
    \clist_gset_eq:NN
    \dim_gset_eq:NN
    \dim_gzero:N
    \dim_gset:Nn
    \dim_gadd:Nn
    \dim_gsub:Nn
    \fp_gset_eq:NN
    \int_gzero:N
    \int_gset_eq:NN
    \int_gadd:Nn
    \int_gsub:Nn
    \int_gincr:N
    \int_gdecr:N
    \int_gset:Nn
    \hbox_gset:Nn
    \hbox_gset_to_wd:Nnn
    \hbox_gset:Nw
    \hbox_gset_to_wd:Nnw
    \muskip_gzero:N
    \muskip_gset:Nn
    \muskip_gadd:Nn
    \muskip_gsub:Nn
    \muskip_gset_eq:NN
    \prop_gclear:N
    \prop_gconcat:NNN
    \prop_gpop:NnN
    \prop_gpop:NnNT
    \prop_gpop:NnNF
    \prop_gpop:NnNTF
    \prop_gput:Nnn
    \prop_gput_if_not_in:Nnn
    \prop_gput_from_keyval:Nn
    \prop_gremove:Nn
    \prop_gset_eq:NN
    \prop_gset_from_keyval:Nn
    \seq_gset_eq:NN
    \skip_gzero:N
    \skip_gset:Nn
    \skip_gset_eq:NN
    \skip_gadd:Nn
```

```
      \skip_gsub:Nn
      \str_gclear:N
      \str_gset_eq:NN
      \str_gput_left:Nn
      \str_gput_right:Nn
      \__kernel_tl_gset:Nx
      \tl_gclear:N
      \tl_gset_eq:NN
      \tl_gput_left:Nn
      \tl_gput_left:NV
      \tl_gput_left:Nv
      \tl_gput_left:Ne
      \tl_gput_left:No
      \tl_gput_right:Nn
      \tl_gput_right:NV
      \tl_gput_right:Nv
      \tl_gput_right:Ne
      \tl_gput_right:No
      \tl_build_gbegin:N
      \tl_build_gput_right:Nn
      \tl_build_gput_left:Nn
      \vbox_gset:Nn
      \vbox_gset_top:Nn
      \vbox_gset_to_ht:Nnn
      \vbox_gset:Nw
      \vbox_gset_to_ht:Nnw
      \vbox_gset_split_to_ht:NNn
    }
```

Scoping for constants.

```
  \__kernel_patch:nnn
    { \__kernel_chk_var_scope:NN c #1 }
    { }
    {
      \bool_const:Nn
      \cctab_const:Nn
      \dim_const:Nn
      \int_const:Nn
      \intarray_const_from_clist:Nn
      \muskip_const:Nn
      \prop_const_from_keyval:Nn
      \prop_const_linked_from_keyval:Nn
      \skip_const:Nn
      \str_const:Nn
      \tl_const:Nn
    }
```

Flag functions.

```
  \__kernel_patch:nnn
    { \__kernel_chk_flag_exist:NN }
    { }
    {
      \flag_ensure_raised:N
      \flag_height:N
      \flag_if_raised:NT
```

```
      \flag_if_raised:NF
      \flag_if_raised:NTF
      \flag_if_raised_p:N
      \flag_raise:N
    }
```

Various one-offs.

```
  \__kernel_patch:nnn
    { \__kernel_chk_cs_exist:N #1 }
    { }
    { \cs_generate_variant:Nn }
  \__kernel_patch:nnn
    { \__kernel_chk_var_scope:NN g #1 }
    { }
    { \cctab_new:N }
  \__kernel_patch:nnn
    { \__kernel_chk_var_scope:NN l #1 }
    { }
    { \flag_new:N }
  \__kernel_patch:nnn
    {
      \__kernel_chk_var_scope:NN l #1
      \__kernel_chk_flag_exist:NN
    }
    { }
    { \flag_clear:N }
  \__kernel_patch:nnn
    { \__kernel_chk_var_scope:NN g #1 }
    { }
    { \intarray_new:Nn }
  \__kernel_patch:nnn
    { \__kernel_chk_var_scope:NN q #1 }
    { }
    { \quark_new:N }
  \__kernel_patch:nnn
    { \__kernel_chk_var_scope:NN s #1 }
    { }
    { \scan_new:N }
```

Patch various internal commands to log definitions of functions. First, a kernel internal. Then internals from the cs, keys and msg modules.

```
  \__kernel_patch:nnn
    { }
    {
      \__kernel_debug_log:e
        { Defining~\token_to_str:N #1~ \msg_line_context: }
    }
    { \__kernel_chk_if_free_cs:N }
⟨@@=cs⟩
  \__kernel_patch_weird:nnn
    {
      \cs_if_free:NF #4
        {
          \__kernel_debug_log:e
            {
```

```
              Variant~\token_to_str:N #4~%
              already~defined;~ not~ changing~ it~ \msg_line_context:
            }
        }
    }
    { }
    { \__cs_generate_variant:wwNN  }
⟨@@=keys⟩
  \__kernel_patch:nnn
    {
      \cs_if_exist:cF { \c__keys_code_root_str #1 }
        { \__kernel_debug_log:e { Defining~key~#1~\msg_line_context: } }
    }
    { }
    { \__keys_cmd_set_direct:nn }
⟨@@=msg⟩
  \__kernel_patch:nnn
    { }
    {
      \__kernel_debug_log:e
        { Defining~message~ #1 / #2 ~\msg_line_context: }
    }
    { \__msg_chk_free:nn }
```

```
⟨@@=prg⟩
```

Internal functions from prg module.

```
  \__kernel_patch_weird:nnn
    { \__kernel_chk_cs_exist:c { #5 _p : #6 } }
    { }
    { \__prg_set_eq_conditional_p_form:wNnnnn }
  \__kernel_patch_weird:nnn
    { \__kernel_chk_cs_exist:c { #5    : #6 TF } }
    { }
    { \__prg_set_eq_conditional_TF_form:wNnnnn }
  \__kernel_patch_weird:nnn
    { \__kernel_chk_cs_exist:c { #5    : #6 T } }
    { }
    { \__prg_set_eq_conditional_T_form:wNnnnn }
  \__kernel_patch_weird:nnn
    { \__kernel_chk_cs_exist:c { #5    : #6 F } }
    { }
    { \__prg_set_eq_conditional_F_form:wNnnnn }
```

```
⟨@@=regex⟩
```

Internal functions from regex module.

```
  \__kernel_patch:nnn
    {
      \__regex_trace_push:nnN { regex } { 1 } \__regex_escape_use:nnnn
      \group_begin:
        \__kernel_tl_set:Nx \l__regex_internal_a_tl
          { \__regex_trace_pop:nnN { regex } { 1 } \__regex_escape_use:nnnn }
        \use_none:nnn
    }
    { }
```

```
    { \__regex_escape_use:nnn }
  \__kernel_patch:nnn
    { \__regex_trace_push:nnN { regex } { 1 } \__regex_build:N }
    {
      \__regex_trace_states:n { 2 }
      \__regex_trace_pop:nnN { regex } { 1 } \__regex_build:N
    }
    { \__regex_build:N }
  \__kernel_patch:nnn
    { \__regex_trace_push:nnN { regex } { 1 } \__regex_build_for_cs:n }
    {
      \__regex_trace_states:n { 2 }
      \__regex_trace_pop:nnN { regex } { 1 } \__regex_build_for_cs:n
    }
    { \__regex_build_for_cs:n }
  \__kernel_patch:nnn
    {
      \__regex_trace:nne { regex } { 2 }
        {
          regex~new~state~
          L=\int_use:N \l__regex_left_state_int ~ -> ~
          R=\int_use:N \l__regex_right_state_int ~ -> ~
          M=\int_use:N \l__regex_max_state_int ~ -> ~
          \int_eval:n { \l__regex_max_state_int + 1 }
        }
    }
    { }
    { \__regex_build_new_state: }
  \__kernel_patch:nnn
    { \__regex_trace_push:nnN { regex } { 1 } \__regex_group_aux:nnnnN }
    { \__regex_trace_pop:nnN { regex } { 1 } \__regex_group_aux:nnnnN }
    { \__regex_group_aux:nnnnN }
  \__kernel_patch:nnn
    { \__regex_trace_push:nnN { regex } { 1 } \__regex_branch:n }
    { \__regex_trace_pop:nnN { regex } { 1 } \__regex_branch:n }
    { \__regex_branch:n }
  \__kernel_patch:nnn
    {
      \__regex_trace_push:nnN { regex } { 1 } \__regex_match:n
      \__regex_trace:nne { regex } { 1 } { analyzing~query~token~list }
    }
    { \__regex_trace_pop:nnN { regex } { 1 } \__regex_match:n }
    { \__regex_match:n }
  \__kernel_patch:nnn
    {
      \__regex_trace_push:nnN { regex } { 1 } \__regex_match_cs:n
      \__regex_trace:nne { regex } { 1 } { analyzing~query~token~list }
    }
    { \__regex_trace_pop:nnN { regex } { 1 } \__regex_match_cs:n }
    { \__regex_match_cs:n }
  \__kernel_patch:nnn
    { \__regex_trace:nne { regex } { 1 } { initializing } }
    { }
    { \__regex_match_init: }
```

```
  \__kernel_patch:nnn
    {
      \__regex_trace:nne { regex } { 2 }
        { state~\int_use:N \l__regex_curr_state_int }
    }
    { }
    { \__regex_use_state: }
  \__kernel_patch:nnn
    { \__regex_trace_push:nnN { regex } { 1 } \__regex_replacement:n }
    { \__regex_trace_pop:nnN { regex } { 1 } \__regex_replacement:n }
    { \__regex_replacement:n }
\group_end:
⟨@@=debug⟩
```

Patching arguments is a bit more involved: we do these one at a time. The basic idea is the same, using a # token that is a string.

```
\group_begin:
  \cs_set_protected:Npn \__kernel_patch:Nn #1
    {
      \group_begin:
        \char_set_catcode_other:N \#
        \__kernel_patch_aux:Nn #1
    }
  \cs_set_protected:Npn \__kernel_patch_aux:Nn #1#2
    {
        \char_set_catcode_parameter:N \#
        \tex_endlinechar:D -1 \scan_stop:
        \exp_args:Ne \tex_scantokens:D
          {
            \tex_global:D \cs_prefix_spec:N #1 \tex_def:D \exp_not:N #1
              \cs_parameter_spec:N #1
              { \exp_args:No \tl_to_str:n { #1 #2 } }
          }
      \group_end:
    }
```

The functions here can get a bit repetitive, so we define a helper which can reuse the same patch code repeatedly. The main part of the patch is the same, so we just have to deal with the part which varies depending on the type of expression.

```
  \cs_set_protected:Npn \__kernel_patch_eval:nn #1#2
    {
      \tl_map_inline:nn {#1}
        {
          \exp_args:NNe \__kernel_patch:Nn ##1
            {
              { \c_hash_str 1 }
              {
                \exp_not:N \__kernel_chk_expr:nNnN { \c_hash_str 2 }
                \exp_not:n {#2}
                \exp_not:N ##1
              }
            }
        }
    }
```

```
⟨@@=dim⟩
  \__kernel_patch_eval:nn
    {
      \dim_set:Nn
      \dim_gset:Nn
      \dim_add:Nn
      \dim_gadd:Nn
      \dim_sub:Nn
      \dim_gsub:Nn
      \dim_const:Nn
    }
    { \__dim_eval:w { } }
⟨@@=int⟩
  \__kernel_patch_eval:nn
    {
      \int_set:Nn
      \int_gset:Nn
      \int_add:Nn
      \int_gadd:Nn
      \int_sub:Nn
      \int_gsub:Nn
      \int_const:Nn
    }
    { \__int_eval:w { } }
  \__kernel_patch_eval:nn
    {
      \muskip_set:Nn
      \muskip_gset:Nn
      \muskip_add:Nn
      \muskip_gadd:Nn
      \muskip_sub:Nn
      \muskip_gsub:Nn
      \muskip_const:Nn
    }
    { \tex_muexpr:D { \tex_mutoglue:D } }
  \__kernel_patch_eval:nn
    {
      \skip_set:Nn
      \skip_gset:Nn
      \skip_add:Nn
      \skip_gadd:Nn
      \skip_sub:Nn
      \skip_gsub:Nn
      \skip_const:Nn
    }
    { \tex_glueexpr:D { } }
```

Patching expandable expressions, first the one-argument versions, then the two-argument ones.

```
  \cs_set_protected:Npn \__kernel_patch_eval:nn #1#2
    {
      \tl_map_inline:nn {#1}
        {
          \exp_args:NNe \__kernel_patch:Nn ##1
```

```
            {
              {
                \exp_not:N \__kernel_chk_expr:nNnN { \c_hash_str 1 }
                \exp_not:n {#2}
                \exp_not:N ##1
              }
            }
        }
    }
⟨@@=box⟩
  \__kernel_patch_eval:nn
    { \__box_dim_eval:n }
    { \__box_dim_eval:w { } }
⟨@@=dim⟩
  \__kernel_patch_eval:nn
    {
      \dim_eval:n
      \dim_to_decimal:n
      \dim_to_decimal_in_sp:n
      \dim_abs:n
      \dim_sign:n
    }
    { \__dim_eval:w { } }
⟨@@=int⟩
  \__kernel_patch_eval:nn
    {
      \int_eval:n
      \int_abs:n
      \int_sign:n
    }
    { \__int_eval:w { } }
  \__kernel_patch_eval:nn
    {
      \skip_eval:n
      \skip_horizontal:n
      \skip_vertical:n
    }
    { \tex_glueexpr:D { } }
  \__kernel_patch_eval:nn
    {
      \muskip_eval:n
    }
    { \tex_muexpr:D { \tex_mutoglue:D } }
  \cs_set_protected:Npn \__kernel_patch_eval:nn #1#2
    {
      \tl_map_inline:nn {#1}
        {
          \exp_args:NNe \__kernel_patch:Nn ##1
            {
              {
                \exp_not:N \__kernel_chk_expr:nNnN { \c_hash_str 1 }
                \exp_not:n {#2}
                \exp_not:N ##1
              }
```

```
              {
                \exp_not:N \__kernel_chk_expr:nNnN { \c_hash_str 2 }
                \exp_not:n {#2}
                \exp_not:N ##1
              }
            }
        }
    }
⟨@@=dim⟩
  \__kernel_patch_eval:nn
    {
      \dim_max:nn
      \dim_min:nn
    }
    { \__dim_eval:w { } }
⟨@@=int⟩
  \__kernel_patch_eval:nn
    {
      \int_max:nn
      \int_min:nn
      \int_div_truncate:nn
      \int_mod:nn
    }
    { \__int_eval:w { } }
```

Conditionals: three argument ones then one argument ones

```
  \cs_set_protected:Npn \__kernel_patch_cond:nn #1#2
    {
      \clist_map_inline:nn { :nNnT , :nNnF , :nNnTF , _p:nNn }
        {
          \exp_args:Nce \__kernel_patch:Nn { #1 ##1 }
            {
              {
                \exp_not:N \__kernel_chk_expr:nNnN { \c_hash_str 1 }
                \exp_not:n {#2}
                \exp_not:c { #1 ##1 }
              }
              { \c_hash_str 2 }
              {
                \exp_not:N \__kernel_chk_expr:nNnN { \c_hash_str 3 }
                \exp_not:n {#2}
                \exp_not:c { #1 ##1 }
              }
            }
        }
    }
⟨@@=dim⟩
  \__kernel_patch_cond:nn { dim_compare } { \__dim_eval:w { } }
⟨@@=int⟩
  \__kernel_patch_cond:nn { int_compare } { \__int_eval:w { } }
  \cs_set_protected:Npn \__kernel_patch_cond:nn #1#2
    {
      \clist_map_inline:nn { :nT , :nF , :nTF , _p:n }
        {
          \exp_args:Nce \__kernel_patch:Nn { #1 ##1 }
```

```
            {
              {
                \exp_not:N \__kernel_chk_expr:nNnN { \c_hash_str 1 }
                \exp_not:n {#2}
                \exp_not:c { #1 ##1 }
              }
            }
        }
    }
⟨@@=int⟩
  \__kernel_patch_cond:nn { int_if_even } { \__int_eval:w { } }
  \__kernel_patch_cond:nn { int_if_odd } { \__int_eval:w { } }
```

Step functions.

```
⟨@@=dim⟩
  \__kernel_patch:Nn \dim_step_function:nnnN
    {
      {
        \__kernel_chk_expr:nNnN {#1} \__dim_eval:w { }
          \dim_step_function:nnnN
      }
      {
        \__kernel_chk_expr:nNnN {#2} \__dim_eval:w { }
          \dim_step_function:nnnN
      }
      {
        \__kernel_chk_expr:nNnN {#3} \__dim_eval:w { }
          \dim_step_function:nnnN
      }
    }
⟨@@=int⟩
  \__kernel_patch:Nn \int_step_function:nnnN
    {
      {
        \__kernel_chk_expr:nNnN {#1} \__int_eval:w { }
          \int_step_function:nnnN
      }
      {
        \__kernel_chk_expr:nNnN {#2} \__int_eval:w { }
          \int_step_function:nnnN
      }
      {
        \__kernel_chk_expr:nNnN {#3} \__int_eval:w { }
          \int_step_function:nnnN
      }
    }
```

Odds and ends

```
  \__kernel_patch:Nn \dim_to_fp:n { { (#1) } }
```

```
\group_end:
```

```
⟨@@=skip⟩
```

This one has catcode changes so must be done by hand.

```
\cs_set_protected:Npn \__skip_tmp:w #1
```

```
  {
    \prg_set_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
      {
        \exp_after:wN \__skip_if_finite:wwNw
        \skip_use:N \tex_glueexpr:D
          \__kernel_chk_expr:nNnN
            {##1} \tex_glueexpr:D { } \skip_if_finite:n
        ; \prg_return_false:
        #1 ; \prg_return_true: \s__skip_stop
      }
  }
\exp_args:No \__skip_tmp:w { \tl_to_str:n { fil } }
⟨@@=msg⟩
```

Messages.

```
\msg_new:nnnn { debug } { debug }
  { The~debugging~option~'#1'~does~not~exist~\msg_line_context:. }
  {
    The~functions~'\iow_char:N\\debug_on:n'~and~
    '\iow_char:N\\debug_off:n'~only~accept~the~arguments~
    'all',~'check-declarations',~'check-expressions',~
    'deprecation',~'log-functions',~not~'#1'.
  }
\msg_new:nnn { debug } { expr } { '#2'~in~#1 }
\msg_new:nnnn { debug } { local-global }
  { Inconsistent~local/global~assignment }
  {
    \c__msg_coding_error_text_tl
    \if:w l #2 Local
    \else:
      \if:w g #2 Global \else: Constant \fi:
    \fi:
    \ %
    assignment~to~a~
    \if:w l #1 local
    \else:
      \if:w g #1 global \else: constant \fi:
    \fi:
    \ %
    variable~'#3'.
  }
\msg_new:nnnn { debug } { non-declared-variable }
  { The~variable~#1~has~not~been~declared~\msg_line_context:. }
  {
    \c__msg_coding_error_text_tl
    Checking~is~active,~and~you~have~tried~do~so~something~like: \\
    \ \ \tl_set:Nn ~ #1 ~ \{ ~ ... ~ \} \\
    without~first~having: \\
    \ \ \tl_new:N ~ #1  \\
    \\
    LaTeX~will~continue,~creating~the~variable~where~it~is~the~one~being~set.
  }
```

__kernel_if_debug:TF Flip the switch for deprecated code.

```
\cs_set_protected:Npn \__kernel_if_debug:TF #1#2 {#1}
```

(*End of definition for* `\__kernel_if_debug:TF`.)

```
⟨/package⟩
```

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

E

F

G

L

Q

S

U

Y