The LaTeX3 Sources

The LaTeX3 Project*

November 21, 2016

Abstract

This is the reference documentation for the expl3 programming environment. The expl3 modules set up an experimental 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.

At present, the expl3 modules are designed to be loaded on top of LaTeX 2_ε. In time, a LaTeX3 format will be produced based on this code. This allows the code to be used in LaTeX 2_ε packages *now* while a stand-alone LaTeX3 is developed.

While expl3 is still experimental, the bundle is now regarded as broadly stable. The syntax conventions and functions provided are now ready for wider use. There may still be changes to some functions, but these will be minor when compared to the scope of expl3.

New modules will be added to the distributed version of expl3 as they reach maturity.

---

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

Contents

Part I
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 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:

D The D specifier means *do not use*. All of the TEX primitives are initially \let to a D name, and some are then given a second name. Only the kernel team should use anything with a D specifier!

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.

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 in general *not* expandable, unless specifically noted.

f The `f` specifier stands for *full expansion*, and in contrast to `x` stops at the first non-expandable item (reading the argument from left to right) without trying to expand it. 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 arbitrary string).

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:

bool Either true or false.

box Box register.

clist Comma separated list.

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

dim "Rigid" lengths.

fp floating-point values;

---

[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.

int Integer-valued count register.

prop Property list.

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

skip "Rubber" lengths.

stream An input or output stream (for reading from or writing to, respectively).

tl Token list variables: placeholder for a token list.

1.1 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 in truth one and the same. 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.

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:

\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 argument (in plain TEX terms, inside an `\edef`), 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:

\xetex_if_engine:*TF* ⋆

`\xetex_if_engine:TF {⟨true code⟩} {⟨false code⟩}`

The underlining and italic of TF indicates that `\xetex_if_engine:T`, `\xetex_if_engine:F` and `\xetex_if_engine:TF` are all available. 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.

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.

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 LaTeX2ε if `\outer` tokens are used in the arguments.

Part II
The **l3bootstrap** package
Bootstrap code

1   Using the LaTeX3 modules

The modules documented in `source3` are designed to be used on top of LaTeX 2$_\varepsilon$ and are loaded all as one with the usual `\usepackage{expl3}` or `\RequirePackage{expl3}` instructions. These modules will also form the basis of the LaTeX3 format, but work in this area is incomplete and not included in this documentation at present.

As the modules use a coding syntax different from standard LaTeX 2$_\varepsilon$ 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 régime in which spaces 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 régime.

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

`\RequirePackage{expl3}`
`\ProvidesExplPackage {`⟨*package*⟩`} {`⟨*date*⟩`} {`⟨*version*⟩`} {`⟨*description*⟩`}`

These functions act broadly in the same way as the corresponding LaTeX 2$_\varepsilon$ 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 2$_\varepsilon$ provides in turning on `\makeatletter` within package and class code.) The ⟨*date*⟩ should be given in the format ⟨*year*⟩`/`⟨*month*⟩`/`⟨*day*⟩.

`\GetIdInfo`

Updated: 2012-06-04

`\RequirePackage{l3bootstrap}`
`\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 alike are loaded with usual LaTeX 2$_\varepsilon$ 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}
```

1.1 Internal functions and variables

`\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`.

Part III
The **l3names** package
Namespace for primitives

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;

- switches to the category code régime for programming;

- provides support settings for building the code as a TeX format.

This module is entirely dedicated to primitives, 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 and LuaTeX should be consulted for details of the primitives. These are named based on the engine which first introduced them:

`\tex_...` Introduced by TeX itself;

`\etex_...` Introduced by the ε-TeX extensions;

`\pdftex_...` Introduced by pdfTeX;

`\xetex_...` Introduced by XeTeX;

`\luatex_...` Introduced by LuaTeX.

Part IV

The **l3basics** package
Basic definitions

As the name suggest this package holds some basic definitions which are needed by most or all other packages in this set.

Here we describe those functions that are used all over the place. With 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.

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.

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.

\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 will be 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). The later will be a } if standard category codes apply.

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" will be fully expanded inside an `x` expansion. In contrast, "protected" functions are not expanded within `x` expansions.

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 will be 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 will result 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 will 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 will 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 `x`-type expansion.

Finally, the functions in Subsections 3.2 and 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 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 2.

3.2 Defining new functions using parameter text

`\cs_new:Npn`
`\cs_new:cpn`
`\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 will result if the ⟨*function*⟩ is already defined.

`\cs_new_nopar:Npn`
`\cs_new_nopar:cpn`
`\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 will result if the ⟨*function*⟩ is already defined.

`\cs_new_protected:Npn`
`\cs_new_protected:cpn`
`\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 x-type argument. The definition is global and an error will result if the ⟨*function*⟩ is already defined.

`\cs_new_protected_nopar:Npn`
`\cs_new_protected_nopar:cpn`
`\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 x-type argument. The definition is global and an error will result if the ⟨*function*⟩ is already defined.

`\cs_set:Npn`
`\cs_set:cpn`
`\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: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: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 x-type argument.

`\cs_set_protected_nopar:Npn`	`\cs_set_protected_nopar:Npn` ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}
`\cs_set_protected_nopar:cpn`	
`\cs_set_protected_nopar:Npx`	
`\cs_set_protected_nopar:cpx`	

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 x-type argument.

`\cs_gset:Npn`	`\cs_gset:Npn` ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}
`\cs_gset:cpn`	
`\cs_gset:Npx`	
`\cs_gset:cpx`	

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:Npn` ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}
`\cs_gset_nopar:cpn`	
`\cs_gset_nopar:Npx`	
`\cs_gset_nopar:cpx`	

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:Npn` ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}
`\cs_gset_protected:cpn`	
`\cs_gset_protected:Npx`	
`\cs_gset_protected:cpx`	

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 x-type argument.

`\cs_gset_protected_nopar:Npn`	`\cs_gset_protected_nopar:Npn` ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}
`\cs_gset_protected_nopar:cpn`	
`\cs_gset_protected_nopar:Npx`	
`\cs_gset_protected_nopar:cpx`	

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 x-type argument.

3.3 Defining new functions using the signature

`\cs_new:Nn`	`\cs_new:Nn` ⟨*function*⟩ {⟨*code*⟩}
`\cs_new:(cn|Nx|cx)`	

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 will result if the ⟨*function*⟩ is already defined.

`\cs_new_nopar:Nn`
`\cs_new_nopar:(cn|Nx|cx)`

`\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 will result if the ⟨*function*⟩ is already defined.

`\cs_new_protected:Nn`
`\cs_new_protected:(cn|Nx|cx)`

`\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 x-type argument. The definition is global and an error will result if the ⟨*function*⟩ is already defined.

`\cs_new_protected_nopar:Nn`
`\cs_new_protected_nopar:(cn|Nx|cx)`

`\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 x-type argument. The definition is global and an error will result if the ⟨*function*⟩ is already defined.

`\cs_set:Nn`
`\cs_set:(cn|Nx|cx)`

`\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|Nx|cx)`

`\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|Nx|cx)`

`\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 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|Nx|cx)

\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 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|Nx|cx)

\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|Nx|cx)

\cs_git cogset_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|Nx|cx)

\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 x-type argument. The assignment of a meaning to the ⟨function⟩ is global.

\cs_gset_protected_nopar:Nn
\cs_gset_protected_nopar:(cn|Nx|cx)

\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 x-type argument. The assignment of a meaning to the ⟨function⟩ is global.

\cs_generate_from_arg_count:NNnn
\cs_generate_from_arg_count:(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.

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 ⟨*cs₁*⟩ ⟨*cs₂*⟩
\cs_new_eq:NN ⟨*cs₁*⟩ ⟨*token*⟩

Globally creates ⟨*control sequence₁*⟩ and sets it to have the same meaning as ⟨*control sequence₂*⟩ 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 ⟨*cs₁*⟩ ⟨*cs₂*⟩
\cs_set_eq:NN ⟨*cs₁*⟩ ⟨*token*⟩

Sets ⟨*control sequence₁*⟩ to have the same meaning as ⟨*control sequence₂*⟩ (or ⟨*token*⟩). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the ⟨*control sequence₁*⟩ is restricted to the current TEX group level.

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

\cs_gset_eq:NN ⟨*cs₁*⟩ ⟨*cs₂*⟩
\cs_gset_eq:NN ⟨*cs₁*⟩ ⟨*token*⟩

Globally sets ⟨*control sequence₁*⟩ to have the same meaning as ⟨*control sequence₂*⟩ (or ⟨*token*⟩). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the ⟨*control sequence₁*⟩ is *not* restricted to the current TEX group level: the assignment is global.

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.

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. This will show the ⟨*replacement text*⟩ for a macro.

TEXhackers note: This is TEX's \meaning primitive. The c variant correctly reports undefined arguments.

15

\cs_show:N
\cs_show:c

Updated: 2015-08-03

`\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.

3.7 Converting to and from control sequences

\use:c ⋆

`\use:c {⟨control sequence name⟩}`

Converts the given ⟨*control sequence name*⟩ into a single control sequence token. This process requires two expansions. 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, they will be of category code 10 (space), 11 (letter) or 12 (other), or a mixture of these.

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:N_TF ⋆
\cs_if_exist_use:c_TF ⋆

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 (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, they will be 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 sequence will *not* include the current escape token, *cf.* `\token_to_str:N`. Full expansion of this function requires exactly 2 expansion steps, and so an x-type expansion, or two o-type expansions will be required to convert the ⟨*control sequence*⟩ to a sequence of characters in the input stream. In most cases, an f-expansion will be correct as well, but this loses a space at the start of the result.

4 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 in absorbing them the outer set will be 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 will absorb between one and four arguments, as indicated by the argument specifier. The braces surrounding each argument will be removed leaving the remaining tokens in the input stream. The category code of these tokens will also be 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 } }
```

will result in the input stream containing

```
abc { def }
```

i.e. only the outer braces will be removed.

`\use_i:nn` ⋆   `\use_i:nn` {⟨*arg₁*⟩} {⟨*arg₂*⟩}
`\use_ii:nn` ⋆

These functions absorb two arguments from the input stream. The function `\use_i:nn` discards the second argument, and leaves the content of the first argument in the input stream. `\use_ii:nn` discards the first argument and leaves the content of the second argument in the input stream. The category code of these tokens will also be fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

\use_i:nnn ⋆	
\use_ii:nnn ⋆	
\use_iii:nnn ⋆	

\use_i:nnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩}

These functions absorb three arguments from the input stream. The function \use_i:nnn discards the second and third arguments, and leaves the content of the first argument in the input stream. \use_ii:nnn and \use_iii:nnn work similarly, leaving the content of second or third arguments in the input stream, respectively. The category code of these tokens will also be fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

\use_i:nnnn ⋆	
\use_ii:nnnn ⋆	
\use_iii:nnnn ⋆	
\use_iv:nnnn ⋆	

\use_i:nnnn {⟨arg₁⟩} {⟨arg₂⟩} {⟨arg₃⟩} {⟨arg₄⟩}

These functions absorb four arguments from the input stream. The function \use_-i:nnnn discards the second, third and fourth arguments, and leaves the content of the first argument in the input stream. \use_ii:nnnn, \use_iii:nnnn and \use_iv:nnnn work similarly, leaving the content of second, third or fourth arguments in the input stream, respectively. The category code of these tokens will also be 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 functions will absorb three arguments and leave the content of the first and second in the input stream. The category code of these tokens will also be fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect. An example:

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

will result in the input stream containing

```
abc { def }
```

i.e. the outer braces will be removed and the third group will be removed.

\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).

\use:x	
Updated: 2011-12-31	

\use:x {⟨expandable tokens⟩}

Fully expands the ⟨*expandable tokens*⟩ and inserts the result into the input stream at the current location. Any hash characters (#) in the argument must be doubled.

4.1 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_nil:w` ⟨*balanced text*⟩ `\q_nil`
`\use_none_delimit_by_q_stop:w`	⋆	`\use_none_delimit_by_q_stop:w` ⟨*balanced text*⟩ `\q_stop`
`\use_none_delimit_by_q_recursion_stop:w`	⋆	`\use_none_delimit_by_q_recursion_stop:w` ⟨*balanced text*⟩ `\q_recursion_stop`

Absorb the ⟨*balanced text*⟩ form 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_nil:nw` {⟨*inserted tokens*⟩} ⟨*balanced text*⟩ `\q_nil`
`\use_i_delimit_by_q_stop:nw`	⋆	`\use_i_delimit_by_q_stop:nw` {⟨*inserted tokens*⟩} ⟨*balanced text*⟩ `\q_stop`
`\use_i_delimit_by_q_recursion_stop:nw`	⋆	`\use_i_delimit_by_q_recursion_stop:nw` {⟨*inserted tokens*⟩} ⟨*balanced text*⟩ `\q_recursion_stop`

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

5 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 will turn 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 carry 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 will usually be 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 will 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 will be 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" will also exist 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 2_ε. 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.

\c_true_bool
\c_false_bool

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

5.1 Tests on control sequences

\cs_if_eq_p:NN ⋆
\cs_if_eq:NN*TF* ⋆

`\cs_if_eq_p:NN {⟨cs₁⟩} {⟨cs₂⟩}`
`\cs_if_eq:NNTF {⟨cs₁⟩} {⟨cs₂⟩} {⟨true code⟩} {⟨false code⟩}`

Compares the definition of two ⟨*control sequences*⟩ and is logically `true` 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 valid definition of ⟨*control sequence*⟩ will evaluate 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 will be `false` if the ⟨*control sequence*⟩ currently exists (as defined by `\cs_if_exist:N`).

5.2 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 will often contain 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 will prefix primitive conditionals with `\if_`.

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

`\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:** These are equivalent to their corresponding TEX primitive conditionals; `\reverse_if:N` is ε-TEX's `\unless`.

`\if_meaning:w` ⋆	`\if_meaning:w` ⟨*arg₁*⟩ ⟨*arg₂*⟩ ⟨*true code*⟩ `\else:` ⟨*false code*⟩ `\fi:`

`\if_meaning:w` executes ⟨*true code*⟩ when ⟨*arg₁*⟩ and ⟨*arg₂*⟩ are the same, otherwise it executes ⟨*false code*⟩. ⟨*arg₁*⟩ and ⟨*arg₂*⟩ could be functions, variables, tokens; in all cases the *unexpanded* definitions are compared.

> **TEXhackers note:** This is TEX's `\ifx`.

`\if:w` ⋆	`\if:w` ⟨*token₁*⟩ ⟨*token₂*⟩ ⟨*true code*⟩ `\else:` ⟨*false code*⟩ `\fi:`
`\if_charcode:w` ⋆	`\if_catcode:w` ⟨*token₁*⟩ ⟨*token₂*⟩ ⟨*true code*⟩ `\else:` ⟨*false code*⟩ `\fi:`
`\if_catcode:w` ⋆	

These conditionals will expand any following tokens until two unexpandable tokens are left. If you wish to prevent this expansion, prefix the token in question with `\exp_not:N`. `\if_catcode:w` tests if the category codes of the two tokens are the same whereas `\if:w` tests if the character codes are identical. `\if_charcode:w` is an alternative name for `\if:w`.

`\if_cs_exist:N` ⋆	`\if_cs_exist:N` ⟨*cs*⟩ ⟨*true code*⟩ `\else:` ⟨*false code*⟩ `\fi:`
`\if_cs_exist:w` ⋆	`\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.

`\if_mode_horizontal:` ⋆	`\if_mode_horizontal:` ⟨*true code*⟩ `\else:` ⟨*false code*⟩ `\fi:`
`\if_mode_vertical:` ⋆	
`\if_mode_math:` ⋆	
`\if_mode_inner:` ⋆	

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

6 Internal kernel functions

`\__chk_if_exist_cs:N`
`\__chk_if_exist_cs:c`

`\__chk_if_exist_cs:N` ⟨*cs*⟩

This function checks that ⟨*cs*⟩ exists according to the criteria for `\cs_if_exist_p:N`, and if not raises a kernel-level error.

`\__chk_if_free_cs:N`
`\__chk_if_free_cs:c`

`\__chk_if_free_cs:N` ⟨*cs*⟩

This function checks that ⟨*cs*⟩ is free according to the criteria for `\cs_if_free_p:N`, and if not raises a kernel-level error.

$\backslash$__chk_if_exist_var:N $\quad$ __chk_if_exist_var:N ⟨var⟩

This function checks that ⟨var⟩ is defined according to the criteria for \cs_if_free_p:N, and if not raises a kernel-level error. This function is only created if the package option check-declarations is active.

$\backslash$__chk_log:x $\quad$ __chk_log:x {⟨message text⟩}

If the log-functions option is active, this function writes the ⟨message text⟩ to the log file using \iow_log:x. Otherwise, the ⟨message text⟩ is ignored using \use_none:n.

$\backslash$__chk_suspend_log: $\quad$ __chk_suspend_log: ... __chk_log:x ... __chk_resume_log:
$\backslash$__chk_resume_log:

Any __chk_log:x command between __chk_suspend_log: and __chk_resume_log: is suppressed. These commands can be nested.

$\backslash$__cs_count_signature:N $\star$ $\quad$ __cs_count_signature:N ⟨function⟩
$\backslash$__cs_count_signature:c $\star$

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.

$\backslash$__cs_split_function:NN $\star$ $\quad$ __cs_split_function:NN ⟨function⟩ ⟨processor⟩

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 after the ⟨processor⟩ function 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⟩ will not include the escape character, and both the ⟨name⟩ and ⟨signature⟩ are made up of tokens with category code 12 (other). The ⟨processor⟩ should be a function with argument specification :nnN (plus any trailing arguments needed).

$\backslash$__cs_get_function_name:N $\star$ $\quad$ __cs_get_function_name:N ⟨function⟩

Splits the ⟨function⟩ into the ⟨name⟩ (*i.e.* the part before the colon) and the ⟨signature⟩ (*i.e.* after the colon). The ⟨name⟩ is then left in the input stream without the escape character present made up of tokens with category code 12 (other).

$\backslash$__cs_get_function_signature:N $\star$ $\quad$ __cs_get_function_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 ⟨signature⟩ is then left in the input stream made up of tokens with category code 12 (other).

$\backslash$__cs_tmp:w $\quad$ 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).

$\backslash$__kernel_register_show:N $\quad$ __kernel_register_show:N ⟨register⟩
$\backslash$__kernel_register_show:c

Used to show the contents of a TEX register at the terminal, formatted such that internal parts of the mechanism are not visible.

$\_\_prg\_case\_end:nw$ ⋆ | $\_\_prg\_case\_end:nw$ {⟨*code*⟩} ⟨*tokens*⟩ \q_mark {⟨*true code*⟩} \q_mark {⟨*false code*⟩}
\q_stop

Used to terminate case statements (\int_case:nnTF, *etc.*) by removing trailing ⟨*tokens*⟩ and the end marker \q_stop, inserting the ⟨*code*⟩ for the successful case (if one is found) and either the `true code` or `false code` for the over all outcome, as appropriate.

Part V

The **l3expan** package
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.

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 from the `\exp_` module. 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` will expand 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 uncritical 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.

2 Methods for defining variants

\cs_generate_variant:Nn

\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*⟩ where these are not already defined. For each ⟨*variant*⟩ given, a function is created which will expand its arguments as detailed and pass 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 }
```

will create a new function `\foo:cn` which will expand its first argument into a control sequence name and pass the result to `\foo:Nn`. Similarly

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

would generate the functions `\foo:NV` and `\foo:cV` in the same way. The `\cs_-generate_variant:Nn` function can only be applied if the ⟨*parent control sequence*⟩ is already defined. Only `n` and `N` arguments can be changed to other types. If the ⟨*parent control sequence*⟩ is protected or if the ⟨*variant*⟩ involves `x` arguments, then the ⟨*variant control sequence*⟩ will also be protected. The ⟨*variant*⟩ is created globally, as is any `\exp_args:N`⟨*variant*⟩ function needed to carry out the expansion.

3 Introducing the variants

The available internal functions for argument expansion come in two flavours, some of them are faster then others. Therefore (when speed is important) it is usually best to follow the following guidelines when defining new functions that are supposed to come with variant forms:

- Arguments that might need expansion should come first in the list of arguments to make processing faster.

- Arguments that should consist of single tokens should come first.

- Arguments that need full expansion (*i.e.*, are denoted with `x`) should be avoided if possible as they can not be processed expandably, *i.e.*, functions of this type will not work correctly in arguments that are themselves subject to `x` expansion.

- In general, unless in the last position, multi-token arguments `n`, `f`, and `o` will need special processing when more than one argument is being expanded. This special processing is not fast. Therefore it is best to use the optimized functions, namely those that contain only `N`, `c`, `V`, and `v`, and, in the last position, `o`, `f`, with possible trailing `N` or `n`, which are not expanded.

The `V` type returns the value of a register, which can be one of `tl`, `int`, `skip`, `dim`, `toks`, 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 `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 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 } }
```

will result in the call `\example:n { 3 , \int_eval:n { 3 + 4 } }` while using `\example:x` instead results in `\example:n { 3 , 7 }` at the cost of being protected. If you use this type of expansion in conditional processing then you should stick to using `TF` type functions only as it does not try to finish any `\if...` `\fi:` itself!

If 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 emphfirst 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.

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: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 will be left unchanged.

`\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. (An internal error will occur if such a conversion is not possible). The result is inserted into the input stream *after* reinsertion of the ⟨*function*⟩. Thus the ⟨*function*⟩ may take more than one argument: all others will be left unchanged.

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

`\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 will be 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. (An internal error will occur if such a conversion is not possible). 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 will be 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 or space is found, 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 will be 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*⟩ second. 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 will be left unchanged.

5 Manipulating two arguments

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

\exp_args:NNc ⟨token₁⟩ ⟨token₂⟩ {⟨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:Nno ⋆	
\exp_args:NnV ⋆	
\exp_args:Nnf ⋆	
\exp_args:Noo ⋆	
\exp_args:Nof ⋆	
\exp_args:Noc ⋆	
\exp_args:Nff ⋆	
\exp_args:Nfo ⋆	
\exp_args:Nnc ⋆	

Updated: 2012-01-14

\exp_args:Noo ⟨token⟩ {⟨tokens₁⟩} {⟨tokens₂⟩}

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 need special (slower) processing.

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

\exp_args:NNx ⟨token₁⟩ ⟨token₂⟩ {⟨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.

6 Manipulating three arguments

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

\exp_args:NNNo ⟨token₁⟩ ⟨token₂⟩ ⟨token₃⟩ {⟨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:NNoo ⋆	
\exp_args:NNno ⋆	
\exp_args:Nnno ⋆	
\exp_args:Nnnc ⋆	
\exp_args:Nooo ⋆	

\exp_args:NNoo ⟨token₁⟩ ⟨token₂⟩ {⟨token₃⟩} {⟨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*. These functions need special (slower) processing.

`\exp_args:NNNx`	`\exp_args:NNnx` ⟨token₁⟩ ⟨token₂⟩ {⟨tokens₁⟩} {⟨tokens₂⟩}
`\exp_args:NNnx`	
`\exp_args:NNox`	
`\exp_args:Nnnx`	
`\exp_args:Nnox`	
`\exp_args:Noox`	
`\exp_args:Ncnx`	
`\exp_args:Nccx`	

New: 2015-08-12

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.*

7 Unbraced expansion

`\exp_last_unbraced:NV`	⋆	`\exp_last_unbraced:Nno` ⟨token⟩ ⟨tokens₁⟩ ⟨tokens₂⟩
`\exp_last_unbraced:(Nf|No|Nv)`	⋆	
`\exp_last_unbraced:Nco`	⋆	
`\exp_last_unbraced:(NcV|NNV|NNo)`	⋆	
`\exp_last_unbraced:Nno`	⋆	
`\exp_last_unbraced:(Noo|Nfo)`	⋆	
`\exp_last_unbraced:NNNV`	⋆	
`\exp_last_unbraced:NNNo`	⋆	
`\exp_last_unbraced:NnNo`	⋆	

Updated: 2012-02-12

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`, and `:Nfo` variants need special (slower) processing.

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

`\exp_last_two_unbraced:Noo` ⋆   `\exp_last_two_unbraced:Noo` ⟨token⟩ ⟨tokens₁⟩ {⟨tokens₂⟩}

This function absorbs three arguments and expand 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 ⟨token₁⟩ ⟨token₂⟩

Carries out a single expansion of ⟨token₂⟩ (which may consume arguments) prior to the expansion of ⟨token₁⟩. If ⟨token₂⟩ is a TEX primitive, it will be executed rather than expanded, while if ⟨token₂⟩ has not expansion (for example, if it is a character) then it will be left unchanged. It is important to notice that ⟨token₁⟩ may be *any* single token, including group-opening and -closing tokens ({ or } assuming normal TEX category codes). Unless specifically required, expansion should be carried out using an appropriate argument specifier variant or the appropriate \exp_arg:N function.

> **TEXhackers note:** This is the TEX primitive \expandafter renamed.

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 will not appear 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 x-type argument.

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

\exp_not:c ⋆ | \exp_not:c {⟨tokens⟩}

Expands the ⟨tokens⟩ until only unexpandable content remains, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited.

\exp_not:n ⋆ | \exp_not:n {⟨tokens⟩}

Prevents expansion of the ⟨tokens⟩ in a context where they would otherwise be expanded, for example an x-type argument.

> **TEXhackers note:** This is the ε-TEX \unexpanded primitive. Hence its argument *must* be surrounded by braces.

\exp_not:V ⋆ | \exp_not:V ⟨variable⟩

Recovers the content of the ⟨variable⟩, then prevents expansion of this material in a context where it would otherwise be expanded, for example an x-type argument.

\exp_not:v ⋆ | \exp_not:v {⟨tokens⟩}

Expands the ⟨tokens⟩ until only unexpandable content 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 is prevented in a context where it would otherwise be expanded, for example an x-type argument.

`\exp_not:o {⟨tokens⟩}`

Expands the ⟨*tokens*⟩ once, then prevents any further expansion in a context where they would otherwise be expanded, for example an `x`-type argument.

`\exp_not:f {⟨tokens⟩}`

Expands ⟨*tokens*⟩ fully until the first unexpandable token is found. Expansion then stops, and the result of the expansion (including any tokens which were not expanded) is protected from further expansion.

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

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:` will terminate the expansion of tokens even if ⟨*more tokens*⟩ are also expandable. The function itself is an implicit space token. Inside an `x`-type expansion, it will retain its form, but when typeset it produces the underlying space (␣).

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. You will find these commands 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 ⟨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.[2]

In typical use cases the `\exp_end:` will be 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.

`\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 will get 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.[3]

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.

`\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 a brace group then the braces are removed. If on the other hand it starts with space tokens then these space tokens are removed while searching for the argument. Thus such space tokens will not terminate the f-type expansion.

10 Internal functions and variables

`\l__exp_internal_tl`

The `\exp_` module has its private variables to temporarily store results of the argument expansion. This is done to avoid interference with other functions using temporary variables.

---

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

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

`\::n`	`\cs_set:Npn \exp_args:Ncof { \::c \::o \::f \::: }`
`\::N`	
`\::p`	Internal forms for the base expansion types. These names do *not* conform to the general
`\::c`	LaTeX3 approach as this makes them more readily visible in the log and so forth.
`\::o`	
`\::f`	
`\::x`	
`\::v`	
`\::V`	
`\:::`	

Part VI
The **l3prg** package
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.

1 Defining a set of conditional functions

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

Updated: 2012-02-06

`\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 will 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_new_protected_conditional:Nnn`
`\prg_set_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 will check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the `set` version will 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.

`\prg_new_eq_conditional:NNn` `\prg_new_eq_conditional:NNn` \⟨*name*₁⟩:⟨*arg spec*₁⟩ \⟨*name*₂⟩:⟨*arg spec*₂⟩
`\prg_set_eq_conditional:NNn` {⟨*conditions*⟩}

These functions copy a family of conditionals. The `new` version will check for existing definitions (*cf.* `\cs_new:Npn`) whereas the `set` version will not (*cf.* `\cs_set:Npn`). 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_true:`
`\prg_return_false:` ⋆	`\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.

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 will generally mean being constructed from predicate functions, 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:N` ⟨*boolean*⟩
`\bool_new:c`	

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

`\bool_set_false:N`	`\bool_set_false:N` ⟨*boolean*⟩
`\bool_set_false:c`	Sets ⟨*boolean*⟩ logically `false`.
`\bool_gset_false:N`	
`\bool_gset_false:c`	

`\bool_set_true:N`	`\bool_set_true:N` ⟨*boolean*⟩
`\bool_set_true:c`	Sets ⟨*boolean*⟩ logically `true`.
`\bool_gset_true:N`	
`\bool_gset_true:c`	

`\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 the content of ⟨*boolean*₁⟩ equal to that of ⟨*boolean*₂⟩.

`\bool_set:Nn`
`\bool_set:cn`
`\bool_gset:Nn`
`\bool_gset:cn`
 	
Updated: 2012-07-08
	`\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_if_p:N` ⋆ `\bool_if_p:c` ⋆ `\bool_if:NTF` ⋆ `\bool_if:cTF` ⋆	`\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_show:N`
`\bool_show:c`
 	
New: 2012-02-09
Updated: 2015-08-01
	`\bool_show:N` ⟨*boolean*⟩
	
	Displays the logical truth of the ⟨*boolean*⟩ on the terminal.

`\bool_show:n` New: 2012-02-09 Updated: 2015-08-07	`\bool_show:n` {⟨*boolean expression*⟩} Displays the logical truth of the ⟨*boolean expression*⟩ on the terminal.

`\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.

`\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.

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 `!` 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.

At present, the infix operators `&&` and `||` perform lazy evaluation as well, but this will change in a future release.

`\bool_if_p:n` ⋆
`\bool_if:nTF` ⋆

Updated: 2012-07-08

`\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_not_p:n` ⋆

Updated: 2012-07-08

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

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

`\bool_xor_p:nn` ⋆

Updated: 2012-07-08

`\bool_xor_p:nn {⟨boolexpr₁⟩} {⟨boolexpr₂⟩}`

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

4 Logical loops

Loops using either boolean expressions or stored boolean values.

`\bool_do_until:Nn` ☆
`\bool_do_until:cn` ☆

`\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*⟩ will be inserted into the input stream again and the process will loop until the ⟨*boolean*⟩ is `true`.

`\bool_do_while:Nn` ☆
`\bool_do_while:cn` ☆

`\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*⟩ will be inserted into the input stream again and the process will loop until the ⟨*boolean*⟩ is `false`.

`\bool_until_do:Nn` ☆
`\bool_until_do:cn` ☆

`\bool_until_do:Nn ⟨boolean⟩ {⟨code⟩}`

This function firsts 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 will then loop until the ⟨*boolean*⟩ is `true`.

`\bool_while_do:Nn` ☆	`\bool_while_do:Nn` ⟨*boolean*⟩ {⟨*code*⟩}
`\bool_while_do:cn` ☆	

This function firsts 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 will then loop until the ⟨*boolean*⟩ is `false`.

`\bool_do_until:nn` ☆	`\bool_do_until:nn` {⟨*boolean expression*⟩} {⟨*code*⟩}
Updated: 2012-07-08	

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*⟩ will be inserted into the input stream again and the process will loop until the ⟨*boolean expression*⟩ evaluates to `true`.

`\bool_do_while:nn` ☆	`\bool_do_while:nn` {⟨*boolean expression*⟩} {⟨*code*⟩}
Updated: 2012-07-08	

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*⟩ will be inserted into the input stream again and the process will loop until the ⟨*boolean expression*⟩ evaluates to `false`.

`\bool_until_do:nn` ☆	`\bool_until_do:nn` {⟨*boolean expression*⟩} {⟨*code*⟩}
Updated: 2012-07-08	

This function firsts 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 will then loop until the ⟨*boolean expression*⟩ is `true`.

`\bool_while_do:nn` ☆	`\bool_while_do:nn` {⟨*boolean expression*⟩} {⟨*code*⟩}
Updated: 2012-07-08	

This function firsts 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 will then loop until the ⟨*boolean expression*⟩ is `false`.

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.

6   Detecting TEX's mode

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

Detects if TEX is currently in horizontal mode.

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

Detects if TEX is currently in inner mode.

`\mode_if_math_p:` ⋆	`\mode_if_math:TF {`⟨*true code*⟩`} {`⟨*false code*⟩`}`
`\mode_if_math:`*TF* ⋆	Detects if TEX is currently in maths mode.
Updated: 2011-09-05	

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

Detects if TEX is currently in vertical mode.

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.

8 Internal programming functions

`\group_align_safe_begin:` ⋆	`\group_align_safe_begin:`
`\group_align_safe_end:` ⋆	`...`
	`\group_align_safe_end:`
Updated: 2011-08-11	

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` will 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.

`\__prg_break_point:Nn` ⋆   `\__prg_break_point:Nn \`⟨*type*⟩`_map_break:`   ⟨*tokens*⟩

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. After the loop ends, the ⟨*tokens*⟩ are 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 ⋆

__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 is simply used as a recognizable marker for the ⟨type⟩.

\g__prg_map_int

This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions __prg_map_1:w, __prg_map_2:w, *etc.*, labelled by \g__prg_-map_int hold functions to be mapped over various list datatypes in inline and variable mappings.

__prg_break_point: ⋆

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

__prg_break: ⋆
__prg_break:n ⋆

__prg_break:n {⟨tokens⟩} ... __prg_break_point:

Breaks a recursion which has no ⟨ending code⟩ and which is not a user-breakable mapping (see for instance \prop_get:Nn), and inserts ⟨tokens⟩ in the input stream.

Part VII
The **l3quark** package
Quarks

1 Introduction to 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_`. Scan marks are for internal use by the kernel: they are not intended for more general use.

1.1 Quarks

Quarks are control sequences 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, with the most command use case as 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. An example of the quark testing functions and their use in recursion can be seen in the implementation of `\clist_map_function:NN`.

2 Defining quarks

`\quark_new:N`

`\quark_new:N` ⟨*quark*⟩

Creates a new ⟨*quark*⟩ which expands only to ⟨*quark*⟩. The ⟨*quark*⟩ will be defined globally, and an error message will be raised if the name was already taken.

\q_stop   Used as a marker for delimited arguments, such as

$$\text{\textbackslash cs\_set:Npn \textbackslash tmp:w \#1\#2 \textbackslash q\_stop \{\#1\}}$$

\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 may 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.

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 later 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).

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 5.

\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 terminates the recursion this is part of using \use_none_delimit_by_q_recursion_stop:w. 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:n {⟨*token list*⟩}
\quark_if_recursion_tail_stop:o

Updated: 2011-09-06

Tests if the ⟨*token list*⟩ contains only \q_recursion_tail, and if so terminates the recursion this is part of using \use_none_delimit_by_q_recursion_stop:w. 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 terminates the recursion this is part of using \use_none_delimit_by_q_recursion_stop:w. 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:nn {⟨*token list*⟩} {⟨*insertion*⟩}
\quark_if_recursion_tail_stop_do:on

Updated: 2011-09-06

Tests if the ⟨*token list*⟩ contains only \q_recursion_tail, and if so terminates the recursion this is part of using \use_none_delimit_by_q_recursion_stop:w. 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.

5 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 will do 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 will overwrite the definition of __my_map_dbl_fn:nn.

6   Internal quark functions

__quark_if_recursion_tail_break:NN   __quark_if_recursion_tail_break:nN {⟨token list⟩}
__quark_if_recursion_tail_break:nN   \⟨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:.

7   Scan marks

Scan marks are control sequences set equal to \scan_stop:, hence will never expand in an expansion context and will be (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).

The scan marks system is only for internal use by the kernel team in a small number of very specific places. These functions should not be used more generally.

__scan_new:N   __scan_new:N ⟨*scan mark*⟩

Creates a new ⟨*scan mark*⟩ which is set equal to \scan_stop:. The ⟨*scan mark*⟩ will be defined globally, and an error message will be raised if the name was already taken by another scan mark.

\s__stop   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

Removes the ⟨*tokens*⟩ and \s__stop from the input stream. This leads to a low-level TeX error if \s__stop is absent.

Part VIII
The **l3token** package
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 will have two primary function categories: \token_ for anything that deals with tokens and \peek_ for looking ahead in the token stream.

Most functions we will 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 will take 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 8.

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 ⟨function⟩. 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:nN {⟨integer expression⟩}`
`\char_set_active_eq:nc`	`\char_gset_active_eq:nN`	⟨function⟩
`\char_gset_active_eq:nc`		

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*⟩. The category code of the ⟨*char*⟩ is *unchanged* by this process. The ⟨*function*⟩ may itself be an active character.

`\char_generate:nn` ⋆

`\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)

- 11 (letter)

- 12 (other)

and other values will raise an error.

The ⟨*charcode*⟩ may be any one valid for the engine in use. Note however that for X$_{\text{Ǝ}}$TEX releases prior to 0.99992 only the 8-bit range (0 to 255) is accepted due to engine limitations.

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
```

`\char_set_catcode_letter:N` ⟨*character*⟩

Updated: 2015-11-11

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
```

`\char_set_catcode_letter:n` {⟨*integer expression*⟩}

Updated: 2015-11-11

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 {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

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 {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

Sets up the behaviour of the ⟨*character*⟩ when found inside `\tl_to_lowercase:n`, such that ⟨*character₁*⟩ will be converted into ⟨*character₂*⟩. 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

Updated: 2015-08-06

`\char_set_uccode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

Sets up the behaviour of the ⟨*character*⟩ when found inside `\tl_to_uppercase:n`, such that ⟨*character₁*⟩ will be converted into ⟨*character₂*⟩. 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

Updated: 2015-08-06

`\char_set_mathcode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

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

Updated: 2015-08-06

`\char_set_sfcode:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

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.

3 Generic tokens

\token_new:Nn

\token_new:Nn ⟨token₁⟩ {⟨token₂⟩}

Defines ⟨token₁⟩ to globally be a snapshot of ⟨token₂⟩. This will be an implicit representation of ⟨token₂⟩.

\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.

\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.

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 will be 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 will be described as macros.

TeXhackers note: This is the TeX primitive \meaning.

\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). The current escape character will be the first character in the sequence, although this will also have 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 renamed.

5 Token conditionals

`\token_if_group_begin_p:N` ⋆
`\token_if_group_begin:NTF` ⋆

`\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:NTF` ⋆

`\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_alignment: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_p:N` ⟨*token*⟩
`\token_if_letter:N`*TF* ⋆	`\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_p:N` ⟨*token*⟩
`\token_if_other:N`*TF* ⋆	`\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_p:N` ⟨*token*⟩
`\token_if_active:N`*TF* ⋆	`\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_p:NN` ⟨*token₁*⟩ ⟨*token₂*⟩
`\token_if_eq_catcode:NN`*TF* ⋆	`\token_if_eq_catcode:NNTF` ⟨*token₁*⟩ ⟨*token₂*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the two ⟨*tokens*⟩ have the same category code.

`\token_if_eq_charcode_p:NN` ⋆	`\token_if_eq_charcode_p:NN` ⟨*token₁*⟩ ⟨*token₂*⟩
`\token_if_eq_charcode:NN`*TF* ⋆	`\token_if_eq_charcode:NNTF` ⟨*token₁*⟩ ⟨*token₂*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the two ⟨*tokens*⟩ have the same character code.

`\token_if_eq_meaning_p:NN` ⋆	`\token_if_eq_meaning_p:NN` ⟨*token₁*⟩ ⟨*token₂*⟩
`\token_if_eq_meaning:NN`*TF* ⋆	`\token_if_eq_meaning:NNTF` ⟨*token₁*⟩ ⟨*token₂*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the two ⟨*tokens*⟩ have the same meaning when expanded.

`\token_if_macro_p:N` ⋆	`\token_if_macro_p:N` ⟨*token*⟩
`\token_if_macro:N`*TF* ⋆	`\token_if_macro:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}
Updated: 2011-05-23	Tests if the ⟨*token*⟩ is a TeX macro.

`\token_if_cs_p:N` ⋆	`\token_if_cs_p:N` ⟨*token*⟩
`\token_if_cs:N`*TF* ⋆	`\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_p:N` ⟨*token*⟩
`\token_if_expandable:N`*TF* ⋆	`\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_p:N` ⟨*token*⟩
`\token_if_long_macro:N`*TF* ⋆	`\token_if_long_macro:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}
Updated: 2012-01-20	Tests if the ⟨*token*⟩ is a long macro.

`\token_if_protected_macro_p:N` ⋆	`\token_if_protected_macro_p:N` ⟨*token*⟩
`\token_if_protected_macro:N`*TF* ⋆	`\token_if_protected_macro:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}
Updated: 2012-01-20	

Tests if the ⟨*token*⟩ is a protected macro: a macro which is both protected and long will return logical `false`.

`\token_if_protected_long_macro_p:N ⋆`	`\token_if_protected_long_macro_p:N ⟨token⟩`
`\token_if_protected_long_macro:NTF ⋆`	`\token_if_protected_long_macro:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}`
Updated: 2012-01-20	

Tests if the ⟨*token*⟩ is a protected long macro.

`\token_if_chardef_p:N ⋆`	`\token_if_chardef_p:N ⟨token⟩`
`\token_if_chardef:NTF ⋆`	`\token_if_chardef:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}`
Updated: 2012-01-20	

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:NTF ⋆`	`\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_dim_register_p:N ⋆`	`\token_if_dim_register_p:N ⟨token⟩`
`\token_if_dim_register:NTF ⋆`	`\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:NTF ⋆`	`\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:NTF ⋆`	`\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:NTF ⋆`	`\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:NTF` ⋆	`\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 byLATEX3).

`\token_if_primitive_p:N` ⋆	`\token_if_primitive_p:N` ⟨*token*⟩
`\token_if_primitive:NTF` ⋆	`\token_if_primitive:NTF` ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}
Updated: 2011-05-23	Tests if the ⟨*token*⟩ is an engine primitive.

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. As peeking ahead does *not* skip spaces the predefined tests include both a space-respecting and space-skipping version.

`\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*⟩ will remain 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*⟩ will remain 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`

`\peek_catcode:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Updated: 2012-12-20

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*⟩ will be left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

\peek_catcode_ignore_spaces:N*TF*

Updated: 2012-12-20

`\peek_catcode_ignore_spaces:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next non-space ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_catcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the ⟨*token*⟩ will be left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

\peek_catcode_remove:N*TF*

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*⟩ will be removed from the input stream if the test is true. The function will then place either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

\peek_catcode_remove_ignore_spaces:N*TF*

Updated: 2012-12-20

`\peek_catcode_remove_ignore_spaces:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next non-space ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_catcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the ⟨*token*⟩ will be removed from the input stream if the test is true. The function will then place either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

\peek_charcode:N*TF*

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*⟩ will be left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

\peek_charcode_ignore_spaces:N*TF*

Updated: 2012-12-20

`\peek_charcode_ignore_spaces:NTF` ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the next non-space ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_charcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the ⟨*token*⟩ will be left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

\peek_charcode_remove:N*TF*

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*⟩ will be removed from the input stream if the test is true. The function will then place either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

\peek_charcode_remove_ignore_spaces:N*TF*

Updated: 2012-12-20

`\peek_charcode_remove_ignore_spaces:NTF` ⟨*test token*⟩
`{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the next non-space ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_charcode:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the ⟨*token*⟩ will be removed from the input stream if the test is true. The function will then place either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

\peek_meaning:N*TF*

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*⟩ will be left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

\peek_meaning_ignore_spaces:N*TF*

Updated: 2012-12-05

`\peek_meaning_ignore_spaces:NTF` ⟨*test token*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the next non-space ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_meaning:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the ⟨*token*⟩ will be left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

\peek_meaning_remove:N*TF*

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*⟩ will be removed from the input stream if the test is true. The function will then place either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

\peek_meaning_remove_ignore_spaces:N*TF*

Updated: 2012-12-05

`\peek_meaning_remove_ignore_spaces:NTF` ⟨*test token*⟩
`{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the next non-space ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_meaning:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the ⟨*token*⟩ will be removed from the input stream if the test is true. The function will then place either the ⟨*true code*⟩ or ⟨*false code*⟩ in the input stream (as appropriate to the result of the test).

7 Decomposing a macro definition

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

\token_get_arg_spec:N ★   \token_get_arg_spec:N ⟨*token*⟩

If the ⟨*token*⟩ is a macro, this function will leave the primitive TeX argument specification in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token \next defined by

    \cs_set:Npn \next #1#2 { x #1 y #2 }

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

TeXhackers note: If the arg spec. contains the string ->, then the spec function will produce incorrect results.

\token_get_replacement_spec:N ★   \token_get_replacement_spec:N ⟨*token*⟩

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

    \cs_set:Npn \next #1#2 { x #1~y #2 }

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

TeXhackers note: If the arg spec. contains the string ->, then the spec function will produce incorrect results.

\token_get_prefix_spec:N ★   \token_get_prefix_spec:N ⟨*token*⟩

If the ⟨*token*⟩ is a macro, this function will leave the TeX prefixes applicable in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token \next defined by

    \cs_set:Npn \next #1#2 { x #1~y #2 }

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

8 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 XƎTeX 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).[4]

There are also a few internal tokens. The following list may be incomplete in some engines.

- Expanding \theþfont 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 both shape and meaning, is inserted when the closing \fi of a conditional is encountered before the conditional is evaluated.

- An \endtemplate (which expands to \outer endtemplate:) can be enountered when peeking ahead at the next token.

- Tricky programming might access a frozen \endwrite.

- Some frozen tokens can only be accessed in interactive sessions: \cr, \right, \endgroup, \fi, \inaccessible.

The meaning of a (non-active) character token is fixed by its category code (and character code) and cannot be changed. We will 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).

---

[4]In LuaTeX, there is also the case of "bytes", which behave as character tokens of category code 12 (other) and character code between 1114112 and 1114366. They are used to output individual bytes to files, rather than UTF-8.

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 will 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 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

⟨*properties*⟩ `macro:`⟨*parameters*⟩`->`⟨*replacement*⟩

where ⟨*properties*⟩ is among `\protected\long\outer`, ⟨*parameters*⟩ describes parameters that the macro expects, such as `#1#2#3`, and ⟨*replacement*⟩ describes how the parameters are manipulated, such as `#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`.

9 Internal functions

`\__char_generate:nn ★`

New: 2016-03-25

`\__char_generate:nn {`⟨*charcode*⟩`} {`⟨*catcode*⟩`}`

This function is identical in operation to the public `\char_generate:nn` but omits various sanity tests. In particular, this means it is used in certain places where engine variations need to be accounted for by the kernel. The ⟨*catcode*⟩ must give an explicit integer after a single expansion.

Part IX
The **l3int** package
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* ("`intexpr`").

1 Integer expressions

`\int_eval:n` ⋆

`\int_eval:n {⟨integer expression⟩}`

Evaluates the ⟨*integer expression*⟩, expanding any integer and token list variables within the ⟨*expression*⟩ to their content (without requiring `\int_use:N`/`\tl_use:N`) and applying the standard mathematical rules. For example both

```
\int_eval: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_eval:n { \l_my_tl +  \l_my_int * 3 - ( 3 + 4 * 5 ) }
```

both evaluate to −6. The {⟨*integer expression*⟩} may contain the operators `+`, `-`, `*` and `/`, along with parenthesis `(` and `)`. Any functions within the expressions should expand to an ⟨*integer denotation*⟩: a sequence of a sign and digits matching the regex `\-?[0-9]+`. After expansion `\int_eval:n` yields an ⟨*integer denotation*⟩ which is left in the input stream.

> **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.

`\int_abs:n` ⋆

Updated: 2012-09-26

`\int_abs:n {⟨integer expression⟩}`

Evaluates the ⟨*integer expression*⟩ 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 {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

Evaluates the two ⟨*integer expressions*⟩ 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 ⟨*integer expression*⟩. 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 {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

Evaluates the two ⟨*integer expressions*⟩ 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 {⟨intexpr₁⟩} {⟨intexpr₂⟩}`
`\int_min:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

Evaluates the ⟨*integer expressions*⟩ 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` ★

Updated: 2012-09-26

`\int_mod:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

Evaluates the two ⟨*integer expressions*⟩ 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 {⟨intexpr₁⟩} {⟨intexpr₂⟩}` times ⟨*intexpr₂*⟩ from ⟨*intexpr₁*⟩. Thus, the result has the same sign as ⟨*intexpr₁*⟩ and its absolute value is strictly less than that of ⟨*intexpr₂*⟩. The result is left in the input stream as an ⟨*integer denotation*⟩ after two expansions.

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*⟩ will initially be equal to 0.

`\int_const:Nn`
`\int_const:cn`

Updated: 2011-10-22

`\int_const:Nn ⟨integer⟩ {⟨integer expression⟩}`

Creates a new constant ⟨*integer*⟩ or raises an error if the name is already taken. The value of the ⟨*integer*⟩ will be set globally to the ⟨*integer expression*⟩.

`\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 ⟨integer₁⟩ ⟨integer₂⟩`

Sets the content of ⟨*integer₁*⟩ equal to that of ⟨*integer₂*⟩.

```
\int_if_exist_p:N ⋆
\int_if_exist_p:c ⋆
\int_if_exist:NTF ⋆
\int_if_exist:cTF ⋆
    New: 2012-03-03
```

`\int_if_exist_p:N` ⟨int⟩
`\int_if_exist:NTF` ⟨int⟩ {⟨true code⟩} {⟨false code⟩}

Tests whether the ⟨int⟩ is currently defined. This does not check that the ⟨int⟩ really is an integer variable.

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⟩ {⟨integer expression⟩}

Adds the result of the ⟨integer expression⟩ 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
\int_gset:Nn
\int_gset:cn
    Updated: 2011-10-22
```

`\int_set:Nn` ⟨integer⟩ {⟨integer expression⟩}

Sets ⟨integer⟩ to the value of ⟨integer expression⟩, 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⟩ {⟨integer expression⟩}

Subtracts the result of the ⟨integer expression⟩ from the current content of the ⟨integer⟩.

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 will be 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.

5 Integer expression conditionals

<div style="text-align:right">

`\int_compare_p:nNn` ⋆
`\int_compare:nNnTF` ⋆

</div>

`\int_compare_p:nNn {⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩}`
`\int_compare:nNnTF`
  `{⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩}`
  `{⟨true code⟩} {⟨false code⟩}`

This function first evaluates each of the ⟨*integer expressions*⟩ as described for `\int_-eval:n`. The two results are then compared using the ⟨*relation*⟩:

Equal	`=`
Greater than	`>`
Less than	`<`

<div style="text-align:right">

`\int_compare_p:n` ⋆
`\int_compare:nTF` ⋆

Updated: 2013-01-13

</div>

`\int_compare_p:n`
  `{`
    `⟨intexpr₁⟩ ⟨relation₁⟩`
    `...`
    `⟨intexpr_N⟩ ⟨relation_N⟩`
    `⟨intexpr_{N+1}⟩`
  `}`
`\int_compare:nTF`
  `{`
    `⟨intexpr₁⟩ ⟨relation₁⟩`
    `...`
    `⟨intexpr_N⟩ ⟨relation_N⟩`
    `⟨intexpr_{N+1}⟩`
  `}`
  `{⟨true code⟩} {⟨false code⟩}`

This function evaluates the ⟨*integer expressions*⟩ as described for `\int_eval:n` and compares consecutive result using the corresponding ⟨*relation*⟩, namely it compares ⟨*intexpr₁*⟩ and ⟨*intexpr₂*⟩ using the ⟨*relation₁*⟩, then ⟨*intexpr₂*⟩ and ⟨*intexpr₃*⟩ using the ⟨*relation₂*⟩, until finally comparing ⟨*intexpr_N*⟩ and ⟨*intexpr_{N+1}*⟩ using the ⟨*relation_N*⟩. The test yields `true` if all comparisons are `true`. Each ⟨*integer expression*⟩ 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	`!=`

`\int_case:nn` ⋆	
`\int_case:nnTF` ⋆	
New: 2013-07-24	

```
\int_case:nnTF {⟨test integer expression⟩}
  {
    {⟨intexpr case₁⟩} {⟨code case₁⟩}
    {⟨intexpr case₂⟩} {⟨code case₂⟩}
    ...
    {⟨intexpr caseₙ⟩} {⟨code caseₙ⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function evaluates the ⟨*test integer expression*⟩ and compares this in turn to each of the ⟨*integer expression cases*⟩. If the two are equal then the associated ⟨*code*⟩ is left in the input stream. 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! }
```

will leave "`Medium`" in the input stream.

`\int_if_even_p:n` ⋆	
`\int_if_even:nTF` ⋆	
`\int_if_odd_p:n` ⋆	
`\int_if_odd:nTF` ⋆	

```
\int_if_odd_p:n {⟨integer expression⟩}
\int_if_odd:nTF {⟨integer expression⟩}
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates the ⟨*integer expression*⟩ as described for `\int_eval:n`. It then evaluates if this is odd or even, as appropriate.

6 Integer expression loops

`\int_do_until:nNnn` ☆

`\int_do_until:nNnn {⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}`

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*integer expressions*⟩ as described for `\int_compare:nNnTF`. If the test is `false` then the ⟨*code*⟩ will be inserted into the input stream again and a loop will occur until the ⟨*relation*⟩ is `true`.

`\int_do_while:nNnn` ☆

`\int_do_while:nNnn {⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}`

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*integer expressions*⟩ as described for `\int_compare:nNnTF`. If the test is `true` then the ⟨*code*⟩ will be inserted into the input stream again and a loop will occur until the ⟨*relation*⟩ is `false`.

`\int_until_do:nNnn` ☆

`\int_until_do:nNnn {⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}`

Evaluates the relationship between the two ⟨*integer expressions*⟩ 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 will be repeated, and a loop will occur until the test is `true`.

`\int_while_do:nNnn` ☆

`\int_while_do:nNnn {⟨intexpr₁⟩} ⟨relation⟩ {⟨intexpr₂⟩} {⟨code⟩}`

Evaluates the relationship between the two ⟨*integer expressions*⟩ 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 will be repeated, and a loop will occur until the test is `false`.

`\int_do_until:nn` ☆

Updated: 2013-01-13

`\int_do_until:nn {⟨integer relation⟩} {⟨code⟩}`

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*⟩ will be inserted into the input stream again and a loop will occur until the ⟨*relation*⟩ is `true`.

`\int_do_while:nn` ☆

Updated: 2013-01-13

`\int_do_while:nn {⟨integer relation⟩} {⟨code⟩}`

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*⟩ will be inserted into the input stream again and a loop will occur until the ⟨*relation*⟩ is `false`.

`\int_until_do:nn` ☆

Updated: 2013-01-13

`\int_until_do:nn {⟨integer relation⟩} {⟨code⟩}`

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 will be repeated, and a loop will occur until the test is `true`.

`\int_while_do:nn` ☆

Updated: 2013-01-13

`\int_while_do:nn {⟨integer relation⟩} {⟨code⟩}`

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 will be repeated, and a loop will occur until the test is `false`.

7 Integer step functions

\int_step_function:nnnN ☆

New: 2012-06-04
Updated: 2014-05-30

\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]

\int_step_inline:nnnn

New: 2012-06-04
Updated: 2014-05-30

\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).

\int_step_variable:nnnNn

New: 2012-06-04
Updated: 2014-05-30

\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*⟩.

8 Formatting integers

Integers can be placed into the output stream with formatting. These conversions apply to any integer expressions.

\int_to_arabic:n ⋆

Updated: 2011-10-22

\int_to_arabic:n {⟨*integer expression*⟩}

Places the value of the ⟨*integer expression*⟩ 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 {⟨integer expression⟩}`

Evaluates the ⟨*integer expression*⟩ 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
  {⟨integer expression⟩} {⟨total symbols⟩}
  ⟨value to symbol mapping⟩
```

This is the low-level function for conversion of an ⟨*integer expression*⟩ into a symbolic form (which will often be 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 {⟨integer expression⟩}`

Calculates the value of the ⟨*integer expression*⟩ 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 {⟨integer expression⟩}`

Calculates the value of the ⟨*integer expression*⟩ 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 {⟨integer expression⟩}`

Calculates the value of the ⟨*integer expression*⟩ 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 {⟨integer expression⟩} {⟨base⟩}`

Calculates the value of the ⟨*integer expression*⟩ 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 {⟨integer expression⟩}`

Places the value of the ⟨*integer expression*⟩ in the input stream as Roman numerals, either lower case (`\int_to_roman:n`) or upper case (`\int_to_Roman:n`). The Roman numerals are letters with category code 11 (letter).

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` ⋆

New: 2014-02-11
Updated: 2014-08-25

`\int_from_hex:n {⟨hexadecimal number⟩}`

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` ⋆

New: 2014-02-11
Updated: 2014-08-25

`\int_from_oct:n {⟨octal number⟩}`

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` ⋆

Updated: 2014-08-25

`\int_from_roman:n {⟨roman numeral⟩}`

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 will be −1. This is the inverse function of `\int_to_roman:n` and `\int_to_Roman:n`.

`\int_from_base:nn` ⋆

Updated: 2014-08-25

`\int_from_base:nn {⟨number⟩} {⟨base⟩}`

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`.

10   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 {⟨integer expression⟩}`

Displays the result of evaluating the ⟨*integer expression*⟩ on the terminal.

71

11   Constant integers

\c_minus_one
\c_zero
\c_one
\c_two
\c_three
\c_four
\c_five
\c_six
\c_seven
\c_eight
\c_nine
\c_ten
\c_eleven
\c_twelve
\c_thirteen
\c_fourteen
\c_fifteen
\c_sixteen
\c_thirty_two
\c_one_hundred
\c_two_hundred_fifty_five
\c_two_hundred_fifty_six
\c_one_thousand
\c_ten_thousand

Integer values used with primitive tests and assignments: 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.

12   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.

13   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: These are both names for 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 (⟨*case₀*⟩) is executed if ⟨*integer*⟩ is 0, the second (⟨*case₁*⟩) 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.

14   Internal functions

__int_to_roman:w ⋆

```
\__int_to_roman:w ⟨integer⟩ ⟨space⟩ or ⟨non-expandable token⟩
```

Converts ⟨*integer*⟩ to it lower case Roman representation. Expansion ends when a space or non-expandable token is found. Note that this function produces a string of letters with category code 12 and that protected functions *are* expanded by this process. Negative ⟨*integer*⟩ values result in no output, although the function does not terminate expansion until a suitable endpoint is found in the same way as for positive numbers.

TEXhackers note: This is the TEX primitive \romannumeral renamed.

__int_value:w ⋆	__int_value:w ⟨*integer*⟩
	__int_value:w ⟨*tokens*⟩ ⟨*optional space*⟩

Expands ⟨*tokens*⟩ until an ⟨*integer*⟩ is formed. One space may be gobbled in the process.

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

__int_eval:w ⋆	__int_eval:w ⟨*intexpr*⟩ __int_eval_end:
__int_eval_end: ⋆	

Evaluates ⟨*integer expression*⟩ as described for \int_eval:n. The evaluation stops when an unexpandable token which is not a valid part of an integer is read or when __int_-eval_end: is reached. The latter is gobbled by the scanner mechanism: __int_eval_-end: itself is unexpandable but used correctly the entire construct is expandable.

> **TEXhackers note:** This is the ε-TEX primitive \numexpr.

__prg_compare_error:	__prg_compare_error:
__prg_compare_error:Nw	__prg_compare_error:Nw ⟨*token*⟩

These are used within \int_compare:nTF, \dim_compare:nTF and so on to recover correctly if the n-type argument does not contain a properly-formed relation.

Part X

The **l3skip** package
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.

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*⟩ will initially be equal to 0 pt.

\dim_const:Nn
\dim_const:cn

New: 2012-03-05

\dim_const:Nn ⟨*dimension*⟩ {⟨*dimension expression*⟩}

Creates a new constant ⟨*dimension*⟩ or raises an error if the name is already taken. The value of the ⟨*dimension*⟩ will be set globally to the ⟨*dimension expression*⟩.

\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:N*TF* ⋆
\dim_if_exist:c*TF* ⋆

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.

2 Setting `dim` variables

```
\dim_add:Nn
\dim_add:cn
\dim_gadd:Nn
\dim_gadd:cn
```

Updated: 2011-10-22

`\dim_add:Nn` ⟨*dimension*⟩ {⟨*dimension* expression⟩}

Adds the result of the ⟨*dimension expression*⟩ to the current content of the ⟨*dimension*⟩.

```
\dim_set:Nn
\dim_set:cn
\dim_gset:Nn
\dim_gset:cn
```

Updated: 2011-10-22

`\dim_set:Nn` ⟨*dimension*⟩ {⟨*dimension* expression⟩}

Sets ⟨*dimension*⟩ to the value of ⟨*dimension expression*⟩, 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*⟩ {⟨*dimension* expression⟩}

Subtracts the result of the ⟨*dimension expression*⟩ from the current content of the ⟨*dimension*⟩.

3 Utilities for dimension calculations

```
\dim_abs:n    ⋆
```

Updated: 2012-09-26

`\dim_abs:n` {⟨*dimexpr*⟩}

Converts the ⟨*dimexpr*⟩ 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` {⟨*dimexpr*$_1$⟩} {⟨*dimexpr*$_2$⟩}
`\dim_min:nn` {⟨*dimexpr*$_1$⟩} {⟨*dimexpr*$_2$⟩}

Evaluates the two ⟨*dimension expressions*⟩ and leaves either the maximum or minimum value in the input stream as appropriate, as a ⟨*dimension denotation*⟩.

\dim_ratio:nn ☆

Updated: 2011-10-22

`\dim_ratio:nn {⟨dimexpr₁⟩} {⟨dimexpr₂⟩}`

Parses the two ⟨*dimension expressions*⟩ and converts the ratio of the two to a form suitable for use inside a ⟨*dimension expression*⟩. 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 ration expression between two integers, with all distances converted to scaled points. Thus

```
\tl_set:Nx \l_my_tl { \dim_ratio:nn { 5 pt } { 10 pt } }
\tl_show:N \l_my_tl
```

will display 327680/655360 on the terminal.

4 Dimension expression conditionals

\dim_compare_p:nNn ⋆
\dim_compare:nNn*TF* ⋆

`\dim_compare_p:nNn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩}`
`\dim_compare:nNnTF`
`  {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩}`
`  {⟨true code⟩} {⟨false code⟩}`

This function first evaluates each of the ⟨*dimension expressions*⟩ as described for `\dim_-eval:n`. The two results are then compared using the ⟨*relation*⟩:

Equal	=
Greater than	>
Less than	<

```
\dim_compare_p:n  ⋆
\dim_compare:nTF  ⋆
```

```
\dim_compare_p:n
  {
     ⟨dimexpr₁⟩ ⟨relation₁⟩
     ...
     ⟨dimexpr_N⟩ ⟨relation_N⟩
     ⟨dimexpr_{N+1}⟩
  }
\dim_compare:nTF
  {
     ⟨dimexpr₁⟩ ⟨relation₁⟩
     ...
     ⟨dimexpr_N⟩ ⟨relation_N⟩
     ⟨dimexpr_{N+1}⟩
  }
  {⟨true code⟩} {⟨false code⟩}
```

This function evaluates the ⟨*dimension expressions*⟩ as described for `\dim_eval:n` and compares consecutive result using the corresponding ⟨*relation*⟩, namely it compares ⟨*dimexpr₁*⟩ and ⟨*dimexpr₂*⟩ using the ⟨*relation₁*⟩, then ⟨*dimexpr₂*⟩ and ⟨*dimexpr₃*⟩ using the ⟨*relation₂*⟩, until finally comparing ⟨*dimexpr_N*⟩ and ⟨*dimexpr_{N+1}*⟩ using the ⟨*relation_N*⟩. The test yields `true` if all comparisons are `true`. Each ⟨*dimension expression*⟩ is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other ⟨*dimension 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	`!=`

`\dim_case:nn` ⋆	
`\dim_case:nnTF` ⋆	
New: 2013-07-24	

```
\dim_case:nnTF {⟨test dimension expression⟩}
  {
     {⟨dimexpr case₁⟩} {⟨code case₁⟩}
     {⟨dimexpr case₂⟩} {⟨code case₂⟩}
     ...
     {⟨dimexpr caseₙ⟩} {⟨code caseₙ⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function evaluates the ⟨*test dimension expression*⟩ and compares this in turn to each of the ⟨*dimension expression cases*⟩. If the two are equal then the associated ⟨*code*⟩ is left in the input stream. 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! }
```

will leave "`Medium`" in the input stream.

5   Dimension expression loops

`\dim_do_until:nNnn` ☆

`\dim_do_until:nNnn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩} {⟨code⟩}`

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*dimension expressions*⟩ as described for `\dim_compare:nNnTF`. If the test is `false` then the ⟨*code*⟩ will be inserted into the input stream again and a loop will occur until the ⟨*relation*⟩ is `true`.

`\dim_do_while:nNnn` ☆

`\dim_do_while:nNnn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩} {⟨code⟩}`

Places the ⟨*code*⟩ in the input stream for TeX to process, and then evaluates the relationship between the two ⟨*dimension expressions*⟩ as described for `\dim_compare:nNnTF`. If the test is `true` then the ⟨*code*⟩ will be inserted into the input stream again and a loop will occur until the ⟨*relation*⟩ is `false`.

`\dim_until_do:nNnn` ☆

`\dim_until_do:nNnn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩} {⟨code⟩}`

Evaluates the relationship between the two ⟨*dimension expressions*⟩ 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 will be repeated, and a loop will occur until the test is `true`.

\dim_while_do:nNnn ☆

\dim_while_do:nNnn {⟨dimexpr₁⟩} ⟨relation⟩ {⟨dimexpr₂⟩} {⟨code⟩}

Evaluates the relationship between the two ⟨*dimension expressions*⟩ 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 will be repeated, and a loop will occur until the test is false.

\dim_do_until:nn ☆

Updated: 2013-01-13

\dim_do_until:nn {⟨dimension relation⟩} {⟨code⟩}

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*⟩ will be inserted into the input stream again and a loop will occur until the ⟨*relation*⟩ is true.

\dim_do_while:nn ☆

Updated: 2013-01-13

\dim_do_while:nn {⟨dimension relation⟩} {⟨code⟩}

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*⟩ will be inserted into the input stream again and a loop will occur until the ⟨*relation*⟩ is false.

\dim_until_do:nn ☆

Updated: 2013-01-13

\dim_until_do:nn {⟨dimension relation⟩} {⟨code⟩}

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 will be repeated, and a loop will occur until the test is true.

\dim_while_do:nn ☆

Updated: 2013-01-13

\dim_while_do:nn {⟨dimension relation⟩} {⟨code⟩}

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 will be repeated, and a loop will occur until the test is false.

6 Using `dim` expressions and variables

\dim_eval:n ⋆

Updated: 2011-10-22

\dim_eval:n {⟨dimension expression⟩}

Evaluates the ⟨*dimension expression*⟩, 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 will be expressed in points (pt), and will require suitable termination if used in a TEX-style assignment as it is *not* an ⟨*internal dimension*⟩.

\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 will be 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.

80

\dim_to_decimal:n ⋆

New: 2014-07-15

\dim_to_decimal:n {⟨dimexpr⟩}

Evaluates the ⟨*dimension expression*⟩, and leaves the result, expressed in points (`pt`) in the input stream, with *no units*. The result is rounded by TeX to four or 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

\dim_to_decimal_in_bp:n {⟨dimexpr⟩}

Evaluates the ⟨*dimension expression*⟩, and leaves the result, expressed in big points (`bp`) in the input stream, with *no units*. The result is rounded by TeX to four or 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.

\dim_to_decimal_in_sp:n ⋆

New: 2015-05-18

\dim_to_decimal_in_sp:n {⟨dimexpr⟩}

Evaluates the ⟨*dimension expression*⟩, and leaves the result, expressed in scaled points (`sp`) in the input stream, with *no units*. The result will necessarily be an integer.

\dim_to_decimal_in_unit:nn ⋆

New: 2014-07-15

\dim_to_decimal_in_unit:nn {⟨dimexpr₁⟩} {⟨dimexpr₂⟩}

Evaluates the ⟨*dimension expressions*⟩, and leaves the value of ⟨*dimexpr₁*⟩, expressed in a unit given by ⟨*dimexpr₂*⟩, in the input stream. The result is a decimal number, rounded by TeX to four or 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_unit:nn { 1bp } { 1mm }
```

leaves `0.35277` in the input stream, *i.e.* the magnitude of one big point when converted to millimetres.

Note that this function is not optimised for any particular output and as such may give different results to \dim_to_decimal_in_bp:n or \dim_to_decimal_in_sp:n. In particular, the latter is able to take a wider range of input values as it is not limited by the ability to calculate a ratio using ε-TeX primitives, which is required internally by \dim_to_decimal_in_unit:nn.

\dim_to_fp:n ⋆

New: 2012-05-08

`\dim_to_fp:n {⟨dimexpr⟩}`

Expands to an internal floating point number equal to the value of the ⟨*dimexpr*⟩ 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 is acceptable.

7 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

New: 2011-11-22
Updated: 2015-08-07

`\dim_show:n {⟨dimension expression⟩}`

Displays the result of evaluating the ⟨*dimension expression*⟩ on the terminal.

8 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.

9 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.

10 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*⟩ will initially be equal to 0 pt.

`\skip_const:Nn`
`\skip_const:cn`
New: 2012-03-05

`\skip_const:Nn` ⟨skip⟩ {⟨skip expression⟩}

Creates a new constant ⟨skip⟩ or raises an error if the name is already taken. The value of the ⟨skip⟩ will be set globally to the ⟨skip expression⟩.

`\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.

11 Setting skip variables

`\skip_add:Nn`
`\skip_add:cn`
`\skip_gadd:Nn`
`\skip_gadd:cn`
Updated: 2011-10-22

`\skip_add:Nn` ⟨skip⟩ {⟨skip expression⟩}

Adds the result of the ⟨skip expression⟩ to the current content of the ⟨skip⟩.

`\skip_set:Nn`
`\skip_set:cn`
`\skip_gset:Nn`
`\skip_gset:cn`
Updated: 2011-10-22

`\skip_set:Nn` ⟨skip⟩ {⟨skip expression⟩}

Sets ⟨skip⟩ to the value of ⟨skip expression⟩, 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` ⟨skip₁⟩ ⟨skip₂⟩

Sets the content of ⟨skip₁⟩ equal to that of ⟨skip₂⟩.

`\skip_sub:Nn`
`\skip_sub:cn`
`\skip_gsub:Nn`
`\skip_gsub:cn`
Updated: 2011-10-22

`\skip_sub:Nn` ⟨skip⟩ {⟨skip expression⟩}

Subtracts the result of the ⟨skip expression⟩ from the current content of the ⟨skip⟩.

12 Skip expression conditionals

`\skip_if_eq_p:nn` *
`\skip_if_eq:nnTF` *

`\skip_if_eq_p:nn {⟨skipexpr₁⟩} {⟨skipexpr₂⟩}`
`\dim_compare:nTF`
`  {⟨skipexpr₁⟩} {⟨skipexpr₂⟩}`
`  {⟨true code⟩} {⟨false code⟩}`

This function first evaluates each of the ⟨*skip expressions*⟩ 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 {⟨skipexpr⟩}`
`\skip_if_finite:nTF {⟨skipexpr⟩} {⟨true code⟩} {⟨false code⟩}`

Evaluates the ⟨*skip expression*⟩ as described for `\skip_eval:n`, and then tests if all of its components are finite.

13 Using `skip` expressions and variables

`\skip_eval:n` *

Updated: 2011-10-22

`\skip_eval:n {⟨skip expression⟩}`

Evaluates the ⟨*skip expression*⟩, 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 will be expressed in points (`pt`), and will require 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 will be 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 `\skip_eval:n`).

TeXhackers note: `\skip_use:N` is the TeX primitive `\the`: this is one of several LaTeX3 names for this primitive.

14 Viewing `skip` variables

`\skip_show:N`
`\skip_show:c`

`\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 expression⟩}`

Displays the result of evaluating the ⟨*skip expression*⟩ on the terminal.

15 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.

16 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.

17 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 {⟨skipexpr⟩}

Inserts a horizontal ⟨skip⟩ into the current list.

TeXhackers note: \skip_horizontal:N is the TeX primitive \hskip renamed.

\skip_vertical:N
\skip_vertical:c
\skip_vertical:n

Updated: 2011-10-22

\skip_vertical:N ⟨skip⟩
\skip_vertical:n {⟨skipexpr⟩}

Inserts a vertical ⟨skip⟩ into the current list.

TeXhackers note: \skip_vertical:N is the TeX primitive \vskip renamed.

18 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⟩ will initially be equal to 0 mu.

`\muskip_const:Nn`
`\muskip_const:cn`
New: 2012-03-05

`\muskip_const:Nn` ⟨*muskip*⟩ {⟨*muskip expression*⟩}

Creates a new constant ⟨*muskip*⟩ or raises an error if the name is already taken. The value of the ⟨*muskip*⟩ will be set globally to the ⟨*muskip expression*⟩.

`\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:NTF` ⋆
`\muskip_if_exist:cTF` ⋆
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.

19 Setting muskip variables

`\muskip_add:Nn`
`\muskip_add:cn`
`\muskip_gadd:Nn`
`\muskip_gadd:cn`
Updated: 2011-10-22

`\muskip_add:Nn` ⟨*muskip*⟩ {⟨*muskip expression*⟩}

Adds the result of the ⟨*muskip expression*⟩ to the current content of the ⟨*muskip*⟩.

`\muskip_set:Nn`
`\muskip_set:cn`
`\muskip_gset:Nn`
`\muskip_gset:cn`
Updated: 2011-10-22

`\muskip_set:Nn` ⟨*muskip*⟩ {⟨*muskip expression*⟩}

Sets ⟨*muskip*⟩ to the value of ⟨*muskip expression*⟩, 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` ⟨*muskip₁*⟩ ⟨*muskip₂*⟩

Sets the content of ⟨*muskip₁*⟩ equal to that of ⟨*muskip₂*⟩.

`\muskip_sub:Nn`
`\muskip_sub:cn`
`\muskip_gsub:Nn`
`\muskip_gsub:cn`
Updated: 2011-10-22

`\muskip_sub:Nn` ⟨*muskip*⟩ {⟨*muskip expression*⟩}

Subtracts the result of the ⟨*muskip expression*⟩ from the current content of the ⟨*skip*⟩.

20 Using `muskip` expressions and variables

`\muskip_eval:n` ⋆

Updated: 2011-10-22

`\muskip_eval:n {`⟨*muskip expression*⟩`}`

Evaluates the ⟨*muskip expression*⟩, 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 will be expressed in `mu`, and will require 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 will be 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.

21 Viewing `muskip` variables

`\muskip_show:N`
`\muskip_show:c`

`\muskip_show:N` ⟨*muskip*⟩

Displays the value of the ⟨*muskip*⟩ on the terminal.

`\muskip_show:n`

New: 2011-11-22
Updated: 2015-08-07

`\muskip_show:n {`⟨*muskip expression*⟩`}`

Displays the result of evaluating the ⟨*muskip expression*⟩ on the terminal.

22 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.

23 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.

24 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`.

25 Internal functions

`\__dim_eval:w` ⋆
`\__dim_eval_end:` ⋆
`\__dim_eval:w ⟨dimexpr⟩ \__dim_eval_end:`

Evaluates ⟨*dimension expression*⟩ as described for `\dim_eval:n`. The evaluation stops when an unexpandable token which is not a valid part of a dimension is read or when `\__dim_eval_end:` is reached. The latter is gobbled by the scanner mechanism: `\__dim_eval_end:` itself is unexpandable but used correctly the entire construct is expandable.

TEXhackers note: This is the ε-TeX primitive `\dimexpr`.

Part XI
The **l3tl** package
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 "token list variable", 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 an manipulate the lists of tokens, and these have the module prefix `tl`. In many cases, function 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.

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*⟩ will initially be empty.

`\tl_const:Nn`
`\tl_const:(Nx|cn|cx)`

`\tl_const:Nn` ⟨*tl var*⟩ {⟨*token list*⟩}

Creates a new constant ⟨*tl var*⟩ or raises an error if the name is already taken. The value of the ⟨*tl var*⟩ will be set globally to the ⟨*token list*⟩.

`\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 var₁*⟩ ⟨*tl var₂*⟩

Sets the content of ⟨*tl var₁*⟩ equal to that of ⟨*tl var₂*⟩.

`\tl_concat:NNN`
`\tl_concat:ccc`
`\tl_gconcat:NNN`
`\tl_gconcat:ccc`
New: 2012-05-18

`\tl_concat:NNN` ⟨*tl var₁*⟩ ⟨*tl var₂*⟩ ⟨*tl var₃*⟩

Concatenates the content of ⟨*tl var₂*⟩ and ⟨*tl var₃*⟩ together and saves the result in ⟨*tl var₁*⟩. The ⟨*tl var₂*⟩ will be placed at the left side of the new token list.

`\tl_if_exist_p:N` ⋆
`\tl_if_exist_p:c` ⋆
`\tl_if_exist:NTF` ⋆
`\tl_if_exist:cTF` ⋆
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.

2 Adding data to token list variables

`\tl_set:Nn`
`\tl_set:(NV|Nv|No|Nf|Nx|cn|cV|cv|co|cf|cx)`
`\tl_gset:Nn`
`\tl_gset:(NV|Nv|No|Nf|Nx|cn|cV|cv|co|cf|cx)`

`\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|No|Nx|cn|cV|co|cx)`
`\tl_gput_left:Nn`
`\tl_gput_left:(NV|No|Nx|cn|cV|co|cx)`

`\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|No|Nx|cn|cV|co|cx)`
`\tl_gput_right:Nn`
`\tl_gput_right:(NV|No|Nx|cn|cV|co|cx)`

`\tl_put_right:Nn` ⟨*tl var*⟩ {⟨*tokens*⟩}

Appends ⟨*tokens*⟩ to the right side of the current content of ⟨*tl var*⟩.

3 Modifying token list variables

`\tl_replace_once:Nnn`
`\tl_replace_once:cnn`
`\tl_greplace_once:Nnn`
`\tl_greplace_once:cnn`

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:cnn`
`\tl_greplace_all:Nnn`
`\tl_greplace_all:cnn`

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:cn`
`\tl_gremove_once:Nn`
`\tl_gremove_once:cn`

Updated: 2011-08-11

`\tl_remove_once:Nn` ⟨*tl var*⟩ {⟨*tokens*⟩}

Removes the first (leftmost) occurrence of ⟨*tokens*⟩ from the ⟨*tl var*⟩. ⟨*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:cn`
`\tl_gremove_all:Nn`
`\tl_gremove_all:cn`

Updated: 2011-08-11

`\tl_remove_all:Nn` ⟨*tl var*⟩ {⟨*tokens*⟩}

Removes all occurrences of ⟨*tokens*⟩ from the ⟨*tl var*⟩. ⟨*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}`

will result in `\l_tmpa_tl` containing `abcd`.

4 Reassigning token list 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).

`\tl_set_rescan:Nnn`	`\tl_set_rescan:Nnn ⟨tl var⟩ {⟨setup⟩} {⟨tokens⟩}`
`\tl_set_rescan:(Nno|Nnx|cnn|cno|cnx)`	
`\tl_gset_rescan:Nnn`	
`\tl_gset_rescan:(Nno|Nnx|cnn|cno|cnx)`	

Updated: 2015-08-11

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⟩ will be 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 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. Only the case of a single line is supported in LuaTEX because of a bug in this engine.

`\tl_rescan:nn`	`\tl_rescan:nn {⟨setup⟩} {⟨tokens⟩}`

Updated: 2015-08-11

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⟩ will be 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 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. Only the case of a single line is supported in LuaTEX because of a bug in this engine.

5 Token list conditionals

`\tl_if_blank_p:n` ⋆	`\tl_if_blank_p:n {⟨token list⟩}`
`\tl_if_blank_p:(V|o)` ⋆	`\tl_if_blank:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}`
`\tl_if_blank:nTF` ⋆	
`\tl_if_blank:(V|o)TF` ⋆	

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:NTF` *
`\tl_if_empty:cTF` *

`\tl_if_empty_p:N` ⟨*tl var*⟩
`\tl_if_empty:NTF` ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list variable*⟩ is entirely empty (*i.e.* contains no tokens at all).

`\tl_if_empty_p:n` *
`\tl_if_empty_p:(V|o)` *
`\tl_if_empty:nTF` *
`\tl_if_empty:(V|o)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:NNTF` *
`\tl_if_eq:(Nc|cN|cc)TF` *

`\tl_if_eq_p:NN` ⟨*tl var₁*⟩ ⟨*tl var₂*⟩
`\tl_if_eq:NNTF` ⟨*tl var₁*⟩ ⟨*tl var₂*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Compares the content of two ⟨*token list variables*⟩ 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:Nx \l_tmpb_tl { \tl_to_str:n { abc } }
\tl_if_eq:NNTF \l_tmpa_tl \l_tmpb_tl { true } { false }
```

yields `false`.

`\tl_if_eq:nnTF`

`\tl_if_eq:nnTF` {⟨*token list₁*⟩} {⟨*token list₂*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token list₁*⟩ and ⟨*token list₂*⟩ contain the same list of tokens, both in respect of character codes and category codes.

`\tl_if_in:NnTF`
`\tl_if_in:cnTF`

`\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|on|no)TF`

`\tl_if_in:nnTF` {⟨*token list₁*⟩} {⟨*token list₂*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token list₂*⟩ is found inside ⟨*token list₁*⟩. The ⟨*token list₂*⟩ 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_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`.

93

`\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_case:Nn` ★
`\tl_case:cn` ★
`\tl_case:NnTF` ★
`\tl_case:cnTF` ★

New: 2013-07-24

```
\tl_case:NnTF ⟨test token list variable⟩
  {
    ⟨token list variable case₁⟩ {⟨code case₁⟩}
    ⟨token list variable case₂⟩ {⟨code case₂⟩}
    ...
    ⟨token list variable caseₙ⟩ {⟨code caseₙ⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function compares the ⟨*test token list variable*⟩ in turn with each of the ⟨*token list variable cases*⟩. If the two are equal (as described for `\tl_if_eq:NNTF`) then the associated ⟨*code*⟩ is left in the input stream. 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 `\tl_case:Nn`, which does nothing if there is no match, is also available.

6 Mapping to token lists

`\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*⟩ will receive 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*⟩ will receive 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 will receive the ⟨*item*⟩ as #1. One in line mapping can be nested inside another. 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 will receive the ⟨*item*⟩ as #1. One in line mapping can be nested inside another. See also `\tl_map_function:nN`.

\tl_map_variable:NNn
\tl_map_variable:cNn

Updated: 2012-06-29

`\tl_map_variable:NNn` ⟨*tl var*⟩ ⟨*variable*⟩ {⟨*function*⟩}

Applies the ⟨*function*⟩ to every ⟨*item*⟩ stored within the ⟨*tl var*⟩. The ⟨*function*⟩ should consist of code which will receive the ⟨*item*⟩ stored in the ⟨*variable*⟩. One variable mapping can be nested inside another. See also `\tl_map_inline:Nn`.

\tl_map_variable:nNn

Updated: 2012-06-29

`\tl_map_variable:nNn` ⟨*token list*⟩ ⟨*variable*⟩ {⟨*function*⟩}

Applies the ⟨*function*⟩ to every ⟨*item*⟩ stored within the ⟨*token list*⟩. The ⟨*function*⟩ should consist of code which will receive the ⟨*item*⟩ stored in the ⟨*variable*⟩. One variable mapping can be nested inside another. See also `\tl_map_inline:nn`.

\tl_map_break: ☆

Updated: 2012-06-29

`\tl_map_break:`

Used to terminate a `\tl_map_...` function before all entries in the ⟨*token list variable*⟩ have been processed. This will normally take 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 will lead to low level TEX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\__prg_break_point:Nn` before the ⟨*tokens*⟩ are inserted into the input stream. This will depend on the design of the mapping function.

\tl_map_break:n ☆

Updated: 2012-06-29

`\tl_map_break:n` {⟨*tokens*⟩}

Used to terminate a `\tl_map_...` function before all entries in the ⟨*token list variable*⟩ have been processed, inserting the ⟨*tokens*⟩ after the mapping has ended. This will normally take place within a conditional statement, for example

```
\tl_map_inline:Nn \l_my_tl
  {
    \str_if_eq:nnT { #1 } { bingo }
      { \tl_map_break:n { <tokens> } }
    % Do something useful
  }
```

Use outside of a `\tl_map_...` scenario will lead to low level TEX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\__prg_break_point:Nn` before the ⟨*tokens*⟩ are inserted into the input stream. This will depend on the design of the mapping function.

7 Using token lists

\tl_to_str:n ⋆

\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).

TEXhackers note: 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 will be 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.

8 Working with the content of token lists

\tl_count:n ⋆
\tl_count:(V|o) ⋆
New: 2012-05-13

\tl_count:n {⟨tokens⟩}

Counts the number of ⟨items⟩ in ⟨tokens⟩ and leaves this information in the input stream. Unbraced tokens count as one element as do each token group ({...}). This process will ignore any unprotected spaces within ⟨tokens⟩. 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 token groups 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 will ignore any unprotected spaces within the ⟨*tl var*⟩. See also \tl_count:n. This function requires three expansions, giving an ⟨*integer denotation*⟩.

\tl_reverse:n *⋆*
\tl_reverse:(V|o) *⋆*
Updated: 2012-01-08

`\tl_reverse:n {⟨token list⟩}`

Reverses the order of the ⟨*items*⟩ in the ⟨*token list*⟩, so that ⟨*item₁*⟩⟨*item₂*⟩⟨*item₃*⟩ ...⟨*itemₙ*⟩ becomes ⟨*itemₙ*⟩...⟨*item₃*⟩⟨*item₂*⟩⟨*item₁*⟩. This process will preserve 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 will not expand further when appearing in an 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⟩`

Reverses the order of the ⟨*items*⟩ stored in ⟨*tl var*⟩, so that ⟨*item₁*⟩⟨*item₂*⟩⟨*item₃*⟩ ...⟨*itemₙ*⟩ becomes ⟨*itemₙ*⟩...⟨*item₃*⟩⟨*item₂*⟩⟨*item₁*⟩. This process will preserve unprotected spaces within the ⟨*token list variable*⟩. Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. See also \tl_reverse:n, and, for improved performance, \tl_reverse_items:n.

\tl_reverse_items:n *⋆*
New: 2012-01-08

`\tl_reverse_items:n {⟨token list⟩}`

Reverses the order of the ⟨*items*⟩ stored in ⟨*tl var*⟩, so that {⟨*item₁*⟩}{⟨*item₂*⟩}{⟨*item₃*⟩} ...{⟨*itemₙ*⟩} becomes {⟨*itemₙ*⟩} ... {⟨*item₃*⟩}{⟨*item₂*⟩}{⟨*item₁*⟩}. This process will remove 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 will not expand further when appearing in an x-type argument expansion.

\tl_trim_spaces:n *⋆*
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 will not expand further when appearing in an x-type argument expansion.

\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⟩`

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the content of the ⟨*tl var*⟩. Note that this therefore *resets* the content of the variable.

97

9   The first token from a token list

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 }`

will both leave `a` in the input stream. If the "head" is a brace group, rather than a single token, the braces will be removed, and so

> `\tl_head:n { ~ { ~ ab } c }`

yields ␣ab. A blank ⟨token list⟩ (see `\tl_if_blank:nTF`) will result 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 will not expand further when appearing in an `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) will result 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 {⟨token list⟩}`
`\tl_tail:n` ⋆	
`\tl_tail:(V|v|f)` ⋆	
Updated: 2012-09-01	

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 }

will both leave ␣{bc}d in the input stream. A blank ⟨*token list*⟩ (see `\tl_if_blank:nTF`) will result 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 will not expand further when appearing in an x-type argument expansion.

`\tl_if_head_eq_catcode_p:nN` ⋆	`\tl_if_head_eq_catcode_p:nN {⟨token list⟩} ⟨test token⟩`
`\tl_if_head_eq_catcode:nNTF` ⋆	`\tl_if_head_eq_catcode:nNTF {⟨token list⟩} ⟨test token⟩`
Updated: 2012-07-09	`{⟨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 will always be `false`.

`\tl_if_head_eq_charcode_p:nN` ⋆	`\tl_if_head_eq_charcode_p:nN {⟨token list⟩} ⟨test token⟩`
`\tl_if_head_eq_charcode_p:fN` ⋆	`\tl_if_head_eq_charcode:nNTF {⟨token list⟩} ⟨test token⟩`
`\tl_if_head_eq_charcode:nNTF` ⋆	`{⟨true code⟩} {⟨false code⟩}`
`\tl_if_head_eq_charcode:fNTF` ⋆	
Updated: 2012-07-09	

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 will always be `false`.

`\tl_if_head_eq_meaning_p:nN` ⋆	`\tl_if_head_eq_meaning_p:nN {⟨token list⟩} ⟨test token⟩`
`\tl_if_head_eq_meaning:nNTF` ⋆	`\tl_if_head_eq_meaning:nNTF {⟨token list⟩} ⟨test token⟩`
Updated: 2012-07-09	`{⟨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 will always be `false`.

`\tl_if_head_is_group_p:n` ⋆	`\tl_if_head_is_group_p:n {⟨token list⟩}`
`\tl_if_head_is_group:nTF` ⋆	`\tl_if_head_is_group:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}`
New: 2012-07-08	

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_p:n {⟨token list⟩}`
`\tl_if_head_is_N_type:nTF` *⋆*	`\tl_if_head_is_N_type:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}`
New: 2012-07-08	

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_p:n {⟨token list⟩}`
`\tl_if_head_is_space:nTF` *⋆*	`\tl_if_head_is_space:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}`
Updated: 2012-07-08	

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ is an explicit space character (explicit token with character code 12 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.

10 Using a single item

`\tl_item:nn` *⋆*	`\tl_item:nn {⟨token list⟩} {⟨integer expression⟩}`
`\tl_item:Nn` *⋆*	
`\tl_item:cn` *⋆*	
New: 2014-07-17	

Indexing items in the ⟨*token list*⟩ from 1 on the left, this function will evaluate the ⟨*integer expression*⟩ and leave 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 thr function expands to nothing.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*item*⟩ will not expand further when appearing in an x-type argument expansion.

11 Viewing token lists

`\tl_show:N`	`\tl_show:N ⟨tl var⟩`
`\tl_show:c`	
Updated: 2015-08-01	

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:n ⟨token list⟩`
Updated: 2015-08-07	

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.

12 Constant token lists

`\c_empty_tl` Constant that is always empty.

`\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.

13 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.

14 Internal functions

`\__tl_trim_spaces:nn` `\__tl_trim_spaces:nn { \q_mark ⟨token list⟩ } {⟨continuation⟩}`

This function removes all leading and trailing explicit space characters from the ⟨*token list*⟩, and expands to the ⟨*continuation*⟩, followed by a brace group containing `\use_-none:n \q_mark` ⟨*trimmed token list*⟩. For instance, `\tl_trim_spaces:n` is implemented by taking the ⟨*continuation*⟩ to be `\exp_not:o`, and the o-type expansion removes the `\q_mark`. This function is also used in l3clist and l3candidates.

Part XII
The **l3str** package
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 will simply be 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 will not treat a token list or the corresponding string representation differently.

Note that as string variables are a special case of token list variables the coverage of `\str_...:N` functions is somewhat smaller than `\tl_...:N`.

The functions `\cs_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N` and `\token_to_str:N` (and variants) will 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.

1   Building strings

`\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*⟩ will initially be empty.

`\str_const:Nn`
`\str_const:(Nx|cn|cx)`

New: 2015-09-18

`\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*⟩ will be 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`
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 var₁*⟩ ⟨*str var₂*⟩

Sets the content of ⟨*str var₁*⟩ equal to that of ⟨*str var₂*⟩.

2 Adding data to string variables

`\str_set:Nn`
`\str_set:(Nx|cn|cx)`
`\str_gset:Nn`
`\str_gset:(Nx|cn|cx)`
New: 2015-09-18

`\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:(Nx|cn|cx)`
`\str_gput_left:Nn`
`\str_gput_left:(Nx|cn|cx)`
New: 2015-09-18

`\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:(Nx|cn|cx)`
`\str_gput_right:Nn`
`\str_gput_right:(Nx|cn|cx)`
New: 2015-09-18

`\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.

2.1 String conditionals

`\str_if_exist_p:N` ⋆
`\str_if_exist_p:c` ⋆
`\str_if_exist:NTF` ⋆
`\str_if_exist:cTF` ⋆
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.

`\str_if_empty_p:N` ⋆
`\str_if_empty_p:c` ⋆
`\str_if_empty:NTF` ⋆
`\str_if_empty:cTF` ⋆
New: 2015-09-18

`\sr_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:NNTF` ⋆
`\str_if_eq:(Nc|cN|cc)TF` ⋆
New: 2015-09-18

`\str_if_eq_p:NN` ⟨*str var₁*⟩ ⟨*str var₂*⟩
`\str_if_eq:NNTF` ⟨*str var₁*⟩ ⟨*str var₂*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Compares the content of two ⟨*str variables*⟩ and is logically `true` if the two contain the same characters.

`\str_if_eq_p:nn` ⋆
`\str_if_eq_p:(Vn|on|no|nV|VV)` ⋆
`\str_if_eq:nnTF` ⋆
`\str_if_eq:(Vn|on|no|nV|VV)TF` ⋆

`\str_if_eq_p:nn` {⟨*tl₁*⟩} {⟨*tl₂*⟩}
`\str_if_eq:nnTF` {⟨*tl₁*⟩} {⟨*tl₂*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Compares the two ⟨*token lists*⟩ on a character by character basis, and is `true` if the two lists 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`.

`\str_if_eq_x_p:nn` ⋆
`\str_if_eq_x:nnTF` ⋆
New: 2012-06-05

`\str_if_eq_x_p:nn` {⟨*tl₁*⟩} {⟨*tl₂*⟩}
`\str_if_eq_x:nnTF` {⟨*tl₁*⟩} {⟨*tl₂*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Compares the full expansion of two ⟨*token lists*⟩ on a character by character basis, and is `true` if the two lists contain the same characters in the same order. Thus for example

`\str_if_eq_x_p:nn { abc } { \tl_to_str:n { abc } }`

is logically `true`.

`\str_case:nn` ⋆
`\str_case:(on|nV|nv)` ⋆
`\str_case:nnTF` ⋆
`\str_case:(on|nV|nv)TF` ⋆
New: 2013-07-24
Updated: 2015-02-28

```
\str_case:nnTF {⟨test string⟩}
  {
    {⟨string case₁⟩} {⟨code case₁⟩}
    {⟨string case₂⟩} {⟨code case₂⟩}
    ...
    {⟨string caseₙ⟩} {⟨code caseₙ⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function compares the ⟨*test string*⟩ in turn with each of the ⟨*string cases*⟩. If the two are equal (as described for `\str_if_eq:nnTF` then the associated ⟨*code*⟩ is left in the input stream. 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.

`\str_case_x:nnTF` ★

New: 2013-07-24

```
\str_case_x:nnTF {⟨test string⟩}
  {
    {⟨string case₁⟩} {⟨code case₁⟩}
    {⟨string case₂⟩} {⟨code case₂⟩}
    ...
    {⟨string caseₙ⟩} {⟨code caseₙ⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function compares the full expansion of the ⟨*test string*⟩ in turn with the full expansion of the ⟨*string cases*⟩. If the two full expansions are equal (as described for `\str_if_eq:nnTF` then the associated ⟨*code*⟩ is left in the input stream. 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_x:nn`, which does nothing if there is no match, is also available. 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.

3 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 will be 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:n {`⟨*token list*⟩`}`
`\str_head:c`	★	
`\str_head:n`	★	
`\str_head_ignore_spaces:n`	★	

New: 2015-09-18

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:n {`⟨*token list*⟩`}`
`\str_tail:c`	★	
`\str_tail:n`	★	
`\str_tail_ignore_spaces:n`	★	

New: 2015-09-18

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` will trim only 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:nn {`⟨*token list*⟩`} {`⟨*integer expression*⟩`}`
`\str_item:nn`	★	
`\str_item_ignore_spaces:nn`	★	

New: 2015-09-18

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:nnn {`⟨*token list*⟩`} {`⟨*start index*⟩`} {`⟨*end index*⟩`}`
`\str_range:cnn`	⋆	
`\str_range:nnn`	⋆	
`\str_range_ignore_spaces:nnn`	⋆	

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and leaves in the input stream the characters from the ⟨*start index*⟩ to the ⟨*end index*⟩ inclusive. Positive ⟨*indices*⟩ are counted from the start of the string, 1 being the first character, and negative ⟨*indices*⟩ are counted from the end of the string, −1 being the last character. If either of ⟨*start index*⟩ or ⟨*end index*⟩ is 0, the result is empty. For instance,

```
\iow_term:x { \str_range:nnn { abcdef } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abcdef } { -4 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { -2 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { 0 } { -1 } }
```

will print `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:x { \str_range:nnn { abcdef } { 5 } { 2 } }
\iow_term:x { \str_range:nnn { abcdef } { -1 } { -4 } }
```

both yield empty strings.

4 String manipulation

`\str_lower_case:n` ☆
`\str_lower_case:f` ☆
`\str_upper_case:n` ☆
`\str_upper_case:f` ☆

New: 2015-03-01

`\str_lower_case:n {`⟨*tokens*⟩`}`
`\str_upper_case: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_upper_case:f { \tl_head:n {#1} }
        \str_lower_case: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_fold_case:n` for this situation (case folding is district from lower casing).

- Case changing text for typesetting: see the `\tl_lower_case:n(n)`, `\tl_upper_-case:n(n)` and `\tl_mixed_case:n(n)` functions which correctly deal with context-dependence and other factors appropriate to text case changing.

TEXhackers note: As with all expl3 functions, the input supported by `\str_fold_case:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with pdfTEX *only* the Latin alphabet characters A–Z will be case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both X⫶TEX and LuaTEX, subject only to the fact that X⫶TEX in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with `\tl_to_str:n`.

`\str_fold_case:n` ⋆
`\str_fold_case:V` ⋆

New: 2014-06-19
Updated: 2016-03-07

`\str_fold_case: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_fold_case: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_fold_case:n` follows the "full" scheme defined by the Unicode Consortium (*e.g.* SSfolds 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).

TEXhackers note: As with all expl3 functions, the input supported by `\str_fold_case:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with pdfTEX *only* the Latin alphabet characters A–Z will be case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both X⅃TEX and LuaTEX, subject only to the fact that X⅃TEX in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with `\tl_to_str:n`.

5 Viewing strings

`\str_show:N`
`\str_show:c`
`\str_show:n`

New: 2015-09-18

`\str_show:N ⟨str var⟩`

Displays the content of the ⟨*str var*⟩ on the terminal.

6 Constant token lists

\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

New: 2015-09-19

Constant strings, containing a single character token, with category code 12.

7 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.

7.1 Internal string functions

__str_if_eq_x:nn ⋆

__str_if_eq_x:nn {⟨tl_1⟩} {⟨tl_2⟩}

Compares the full expansion of two ⟨*token lists*⟩ on a character by character basis, and is true if the two lists contain the same characters in the same order. Leaves 0 in the input stream if the condition is true, and +1 or -1 otherwise.

__str_if_eq_x_return:nn

__str_if_eq_x_return:nn {⟨tl_1⟩} {⟨tl_2⟩}

Compares the full expansion of two ⟨*token lists*⟩ on a character by character basis, and is true if the two lists contain the same characters in the same order. Either \prg_-return_true: or \prg_return_false: is then left in the input stream. This is a version of \str_if_eq_x:nnTF coded for speed.

__str_to_other:n ⋆

__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.

`\__str_count:n` ⋆   `\__str_count:n {⟨other string⟩}`

This function expects an argument that is entirely made of characters with category "other", as produced by `\__str_to_other:n`. It leaves in the input stream the number of character tokens in the ⟨*other string*⟩, faster than the analogous `\str_count:n` function.

`\__str_range:nnn` ⋆   `\__str_range:nnn {⟨other string⟩} {⟨start index⟩} {⟨end index⟩}`

Identical to `\str_range:nnn` except that the first argument is expected to be entirely made of characters with category "other", as produced by `\__str_to_other:n`, and the result is also an ⟨*other string*⟩.

Part XIII

The **l3seq** package
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.

1 Creating and initialising sequences

\seq_new:N
\seq_new:c

\seq_new:N ⟨*sequence*⟩

Creates a new ⟨*sequence*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*sequence*⟩ will initially contain no items.

\seq_clear:N
\seq_clear:c
\seq_gclear:N
\seq_gclear:c

\seq_clear:N ⟨*sequence*⟩

Clears all items from the ⟨*sequence*⟩.

\seq_clear_new:N
\seq_clear_new:c
\seq_gclear_new:N
\seq_gclear_new:c

\seq_clear_new:N ⟨*sequence*⟩

Ensures that the ⟨*sequence*⟩ exists globally by applying \seq_new:N if necessary, then applies \seq_(g)clear:N to leave the ⟨*sequence*⟩ 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 ⟨*sequence₁*⟩ ⟨*sequence₂*⟩

Sets the content of ⟨*sequence₁*⟩ equal to that of ⟨*sequence₂*⟩.

\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

\seq_set_from_clist:NN ⟨*sequence*⟩ ⟨*comma-list*⟩

New: 2014-07-17

Converts the data in the ⟨*comma list*⟩ into a ⟨*sequence*⟩: the original ⟨*comma list*⟩ is unchanged.

```
\seq_set_split:Nnn
\seq_set_split:NnV
\seq_gset_split:Nnn
\seq_gset_split:NnV
```
New: 2011-08-15
Updated: 2012-07-02

`\seq_set_split:Nnn ⟨sequence⟩ {⟨delimiter⟩} {⟨token list⟩}`

Splits the ⟨*token list*⟩ into ⟨*items*⟩ separated by ⟨*delimiter*⟩, and assigns the result to the ⟨*sequence*⟩. 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` ⟨*sequence*⟩ {⟨⟩}. 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*⟩.

```
\seq_concat:NNN
\seq_concat:ccc
\seq_gconcat:NNN
\seq_gconcat:ccc
```

`\seq_concat:NNN ⟨sequence₁⟩ ⟨sequence₂⟩ ⟨sequence₃⟩`

Concatenates the content of ⟨*sequence₂*⟩ and ⟨*sequence₃*⟩ together and saves the result in ⟨*sequence₁*⟩. The items in ⟨*sequence₂*⟩ will be placed at the left side of the new sequence.

```
\seq_if_exist_p:N  ⋆
\seq_if_exist_p:c  ⋆
\seq_if_exist:NTF  ⋆
\seq_if_exist:cTF  ⋆
```
New: 2012-03-03

`\seq_if_exist_p:N ⟨sequence⟩`
`\seq_if_exist:NTF ⟨sequence⟩ {⟨true code⟩} {⟨false code⟩}`

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

2 Appending data to sequences

```
\seq_put_left:Nn
\seq_put_left:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
\seq_gput_left:Nn
\seq_gput_left:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
```

`\seq_put_left:Nn ⟨sequence⟩ {⟨item⟩}`

Appends the ⟨*item*⟩ to the left of the ⟨*sequence*⟩.

```
\seq_put_right:Nn
\seq_put_right:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
\seq_gput_right:Nn
\seq_gput_right:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
```

`\seq_put_right:Nn ⟨sequence⟩ {⟨item⟩}`

Appends the ⟨*item*⟩ to the right of the ⟨*sequence*⟩.

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 ⟨*token list variable*⟩ 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 ⟨sequence⟩ ⟨token list variable⟩`

Stores the left-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩ without removing it from the ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally. If ⟨*sequence*⟩ is empty the ⟨*token list variable*⟩ will contain the special marker `\q_no_value`.

\seq_get_right:NN
\seq_get_right:cN

Updated: 2012-05-19

\seq_get_right:NN ⟨sequence⟩ ⟨token list variable⟩

Stores the right-most item from a ⟨sequence⟩ in the ⟨token list variable⟩ without removing it from the ⟨sequence⟩. The ⟨token list variable⟩ is assigned locally. If ⟨sequence⟩ is empty the ⟨token list variable⟩ will contain the special marker \q_no_value.

\seq_pop_left:NN
\seq_pop_left:cN

Updated: 2012-05-14

\seq_pop_left:NN ⟨sequence⟩ ⟨token list variable⟩

Pops the left-most item from a ⟨sequence⟩ into the ⟨token list variable⟩, *i.e.* removes the item from the sequence and stores it in the ⟨token list variable⟩. Both of the variables are assigned locally. If ⟨sequence⟩ is empty the ⟨token list variable⟩ will contain the special marker \q_no_value.

\seq_gpop_left:NN
\seq_gpop_left:cN

Updated: 2012-05-14

\seq_gpop_left:NN ⟨sequence⟩ ⟨token list variable⟩

Pops the left-most item from a ⟨sequence⟩ into the ⟨token list variable⟩, *i.e.* removes the item from the sequence and stores it in the ⟨token list variable⟩. The ⟨sequence⟩ is modified globally, while the assignment of the ⟨token list variable⟩ is local. If ⟨sequence⟩ is empty the ⟨token list variable⟩ will contain the special marker \q_no_value.

\seq_pop_right:NN
\seq_pop_right:cN

Updated: 2012-05-19

\seq_pop_right:NN ⟨sequence⟩ ⟨token list variable⟩

Pops the right-most item from a ⟨sequence⟩ into the ⟨token list variable⟩, *i.e.* removes the item from the sequence and stores it in the ⟨token list variable⟩. Both of the variables are assigned locally. If ⟨sequence⟩ is empty the ⟨token list variable⟩ will contain the special marker \q_no_value.

\seq_gpop_right:NN
\seq_gpop_right:cN

Updated: 2012-05-19

\seq_gpop_right:NN ⟨sequence⟩ ⟨token list variable⟩

Pops the right-most item from a ⟨sequence⟩ into the ⟨token list variable⟩, *i.e.* removes the item from the sequence and stores it in the ⟨token list variable⟩. The ⟨sequence⟩ is modified globally, while the assignment of the ⟨token list variable⟩ is local. If ⟨sequence⟩ is empty the ⟨token list variable⟩ will contain the special marker \q_no_value.

\seq_item:Nn ⋆
\seq_item:cn ⋆

New: 2014-07-17

\seq_item:Nn ⟨sequence⟩ {⟨integer expression⟩}

Indexing items in the ⟨sequence⟩ from 1 at the top (left), this function will evaluate the ⟨integer expression⟩ and leave 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. When the ⟨integer expression⟩ is larger than the number of items in the ⟨sequence⟩ (as calculated by \seq_count:N) then the function will expand to nothing.

> **TEXhackers note:** The result is returned within the \unexpanded primitive (\exp_not:n), which means that the ⟨item⟩ will not expand further when appearing in an x-type argument expansion.

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:NN_TF
\seq_get_left:cN_TF

New: 2012-05-14
Updated: 2012-05-19

`\seq_get_left:NNTF` ⟨*sequence*⟩ ⟨*token list variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, stores the left-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩ without removing it from a ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally.

\seq_get_right:NN_TF
\seq_get_right:cN_TF

New: 2012-05-19

`\seq_get_right:NNTF` ⟨*sequence*⟩ ⟨*token list variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, stores the right-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩ without removing it from a ⟨*sequence*⟩. The ⟨*token list variable*⟩ is assigned locally.

\seq_pop_left:NN_TF
\seq_pop_left:cN_TF

New: 2012-05-14
Updated: 2012-05-19

`\seq_pop_left:NNTF` ⟨*sequence*⟩ ⟨*token list variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, pops the left-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from a ⟨*sequence*⟩. Both the ⟨*sequence*⟩ and the ⟨*token list variable*⟩ are assigned locally.

\seq_gpop_left:NN_TF
\seq_gpop_left:cN_TF

New: 2012-05-14
Updated: 2012-05-19

`\seq_gpop_left:NNTF` ⟨*sequence*⟩ ⟨*token list variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, pops the left-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from a ⟨*sequence*⟩. The ⟨*sequence*⟩ is modified globally, while the ⟨*token list variable*⟩ is assigned locally.

\seq_pop_right:NN_TF
\seq_pop_right:cN_TF

New: 2012-05-19

`\seq_pop_right:NNTF` ⟨*sequence*⟩ ⟨*token list variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, pops the right-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from a ⟨*sequence*⟩. Both the ⟨*sequence*⟩ and the ⟨*token list variable*⟩ are assigned locally.

\seq_gpop_right:NN_TF
\seq_gpop_right:cN_TF

New: 2012-05-19

`\seq_gpop_right:NNTF` ⟨*sequence*⟩ ⟨*token list variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*sequence*⟩ is empty, leaves the ⟨*false code*⟩ in the input stream. The value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon. If the ⟨*sequence*⟩ is non-empty, pops the right-most item from a ⟨*sequence*⟩ in the ⟨*token list variable*⟩, *i.e.* removes the item from a ⟨*sequence*⟩. The ⟨*sequence*⟩ is modified globally, while the ⟨*token list variable*⟩ is assigned locally.

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` ⟨*sequence*⟩

Removes duplicate items from the ⟨*sequence*⟩, leaving the left most copy of each item in the ⟨*sequence*⟩. 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 ⟨*sequence*⟩ and does a comparison with the ⟨*items*⟩ already checked. It is therefore relatively slow with large sequences.

`\seq_remove_all:Nn`
`\seq_remove_all:cn`
`\seq_gremove_all:Nn`
`\seq_gremove_all:cn`

`\seq_remove_all:Nn` ⟨*sequence*⟩ {⟨*item*⟩}

Removes every occurrence of ⟨*item*⟩ from the ⟨*sequence*⟩. The ⟨*item*⟩ comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

`\seq_reverse:N`
`\seq_reverse:c`
`\seq_greverse:N`
`\seq_greverse:c`

New: 2014-07-18

`\seq_reverse:N` ⟨*sequence*⟩

Reverses the order of the items stored in the ⟨*sequence*⟩.

6 Sequence conditionals

`\seq_if_empty_p:N` ⋆
`\seq_if_empty_p:c` ⋆
`\seq_if_empty:NTF` ⋆
`\seq_if_empty:cTF` ⋆

`\seq_if_empty_p:N` ⟨*sequence*⟩
`\seq_if_empty:NTF` ⟨*sequence*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*sequence*⟩ is empty (containing no items).

`\seq_if_in:NnTF`
`\seq_if_in:(NV|Nv|No|Nx|cn|cV|cv|co|cx)TF`

`\seq_if_in:NnTF` ⟨*sequence*⟩ {⟨*item*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*item*⟩ is present in the ⟨*sequence*⟩.

7 Mapping to sequences

`\seq_map_function:NN` ☆
`\seq_map_function:cN` ☆

Updated: 2012-06-29

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

Applies ⟨*function*⟩ to every ⟨*item*⟩ stored in the ⟨*sequence*⟩. The ⟨*function*⟩ will receive one argument for each iteration. The ⟨*items*⟩ are returned from left to right. The function `\seq_map_inline:Nn` is faster than `\seq_map_function:NN` for sequences with more than about 10 items. One mapping may be nested inside another.

`\seq_map_inline:Nn`
`\seq_map_inline:cn`

Updated: 2012-06-29

`\seq_map_inline:Nn` ⟨*sequence*⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*sequence*⟩. The ⟨*inline function*⟩ should consist of code which will receive the ⟨*item*⟩ as `#1`. One in line mapping can be nested inside another. The ⟨*items*⟩ are returned from left to right.

`\seq_map_variable:NNn`
`\seq_map_variable:`(Ncn|cNn|ccn)

Updated: 2012-06-29

`\seq_map_variable:NNn` ⟨sequence⟩ ⟨tl var.⟩ {⟨function using tl var.⟩}

Stores each entry in the ⟨sequence⟩ in turn in the ⟨tl var.⟩ and applies the ⟨function using tl var.⟩ The ⟨function⟩ will usually consist of code making use of the ⟨tl var.⟩, but this is not enforced. One variable mapping can be nested inside another. The ⟨items⟩ are returned from left to right.

`\seq_map_break:` ☆

Updated: 2012-06-29

`\seq_map_break:`

Used to terminate a `\seq_map_...` function before all entries in the ⟨sequence⟩ have been processed. This will normally take 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 will lead to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\__prg_break_point:Nn` before further items are taken from the input stream. This will depend on the design of the mapping function.

`\seq_map_break:n` ☆

Updated: 2012-06-29

`\seq_map_break:n` {⟨tokens⟩}

Used to terminate a `\seq_map_...` function before all entries in the ⟨sequence⟩ have been processed, inserting the ⟨tokens⟩ after the mapping has ended. This will normally take place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \seq_map_break:n { <tokens> } }
      {
        % Do something useful
      }
  }
```

Use outside of a `\seq_map_...` scenario will lead to low level TeX errors.

TeXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\__prg_break_point:Nn` before the ⟨tokens⟩ are inserted into the input stream. This will depend on the design of the mapping function.

\seq_count:N ★
\seq_count:c ★

New: 2012-07-13

`\seq_count:N ⟨sequence⟩`

Leaves the number of items in the ⟨*sequence*⟩ in the input stream as an ⟨*integer denotation*⟩. The total number of items in a ⟨*sequence*⟩ will include those which are empty and duplicates, *i.e.* every item in a ⟨*sequence*⟩ is unique.

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 will be 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~ }
```

will insert "`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*⟩ will not expand further when appearing in an 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 will be 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~ }
```

will insert "`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*⟩ will not expand further when appearing in an x-type argument expansion.

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` ⟨sequence⟩ ⟨token list variable⟩

Reads the top item from a ⟨sequence⟩ into the ⟨token list variable⟩ without removing it from the ⟨sequence⟩. The ⟨token list variable⟩ is assigned locally. If ⟨sequence⟩ is empty the ⟨token list variable⟩ will contain the special marker `\q_no_value`.

`\seq_pop:NN`
`\seq_pop:cN`

Updated: 2012-05-14

`\seq_pop:NN` ⟨sequence⟩ ⟨token list variable⟩

Pops the top item from a ⟨sequence⟩ into the ⟨token list variable⟩. Both of the variables are assigned locally. If ⟨sequence⟩ is empty the ⟨token list variable⟩ will contain the special marker `\q_no_value`.

`\seq_gpop:NN`
`\seq_gpop:cN`

Updated: 2012-05-14

`\seq_gpop:NN` ⟨sequence⟩ ⟨token list variable⟩

Pops the top item from a ⟨sequence⟩ into the ⟨token list variable⟩. The ⟨sequence⟩ is modified globally, while the ⟨token list variable⟩ is assigned locally. If ⟨sequence⟩ is empty the ⟨token list variable⟩ will contain the special marker `\q_no_value`.

`\seq_get:NNTF`
`\seq_get:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_get:NNTF` ⟨sequence⟩ ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}

If the ⟨sequence⟩ is empty, leaves the ⟨false code⟩ in the input stream. The value of the ⟨token list variable⟩ is not defined in this case and should not be relied upon. If the ⟨sequence⟩ is non-empty, stores the top item from a ⟨sequence⟩ in the ⟨token list variable⟩ without removing it from the ⟨sequence⟩. The ⟨token list variable⟩ is assigned locally.

`\seq_pop:NNTF`
`\seq_pop:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_pop:NNTF` ⟨sequence⟩ ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}

If the ⟨sequence⟩ is empty, leaves the ⟨false code⟩ in the input stream. The value of the ⟨token list variable⟩ is not defined in this case and should not be relied upon. If the ⟨sequence⟩ is non-empty, pops the top item from the ⟨sequence⟩ in the ⟨token list variable⟩, *i.e.* removes the item from the ⟨sequence⟩. Both the ⟨sequence⟩ and the ⟨token list variable⟩ are assigned locally.

`\seq_gpop:NNTF`
`\seq_gpop:cNTF`

New: 2012-05-14
Updated: 2012-05-19

`\seq_gpop:NNTF` ⟨sequence⟩ ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}

If the ⟨sequence⟩ is empty, leaves the ⟨false code⟩ in the input stream. The value of the ⟨token list variable⟩ is not defined in this case and should not be relied upon. If the ⟨sequence⟩ is non-empty, pops the top item from the ⟨sequence⟩ in the ⟨token list variable⟩, *i.e.* removes the item from the ⟨sequence⟩. The ⟨sequence⟩ is modified globally, while the ⟨token list variable⟩ is assigned locally.

`\seq_push:Nn`
`\seq_push:(NV|Nv|No|Nx|cn|cV|cv|co|cx)`
`\seq_gpush:Nn`
`\seq_gpush:(NV|Nv|No|Nx|cn|cV|cv|co|cx)`

`\seq_push:Nn` ⟨sequence⟩ {⟨item⟩}

Adds the {⟨item⟩} to the top of the ⟨sequence⟩.

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 occurences of a given item. To make sure that a ⟨*sequence variable*⟩ only has distinct items, use `\seq_remove_duplicates:N` ⟨*sequence variable*⟩. 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
```

11   Constant and scratch sequences

`\c_empty_seq`

New: 2012-07-02

Constant that is always empty.

`\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.

12   Viewing sequences

`\seq_show:N`
`\seq_show:c`

Updated: 2015-08-01

`\seq_show:N` ⟨*sequence*⟩

Displays the entries in the ⟨*sequence*⟩ in the terminal.

13 Internal sequence functions

\s__seq   This scan mark (equal to \scan_stop:) marks the beginning of a sequence variable.

__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 will be raised. The definition should always be set globally.

__seq_push_item_def:n
__seq_push_item_def:x   __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.

Part XIV

The **l3clist** package
Comma separated lists

Comma lists contain ordered data where items can be added to the left or right end of the list. The resulting ordered list can then be mapped over using `\clist_map_function:NN`. Several items can be added at once, and spaces are removed from both sides of each item on input. Hence,

```
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { ~ a ~ , ~ {b} ~ }
\clist_put_right:Nn \l_my_clist { ~ { c ~ } , d }
```

results in `\l_my_clist` containing `a,{b},{c~},d`. Comma lists cannot contain empty items, thus

```
\clist_clear_new:N \l_my_clist
\clist_put_right:Nn \l_my_clist { , ~ , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
```

will leave `true` in the input stream. To include an item which contains a comma, or starts or ends with a space, surround it with braces. The sequence data type should be preferred to comma lists if items are to contain `{`, `}`, or `#` (assuming the usual TEX category codes apply).

1 Creating and initialising comma lists

`\clist_new:N`
`\clist_new:c`

`\clist_new:N` ⟨*comma list*⟩

Creates a new ⟨*comma list*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*comma list*⟩ will initially contain no items.

`\clist_const:Nn`
`\clist_const:(Nx|cn|cx)`

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*⟩ will be set globally to the ⟨*comma list*⟩.

`\clist_clear:N`
`\clist_clear:c`
`\clist_gclear:N`
`\clist_gclear:c`

`\clist_clear:N` ⟨*comma list*⟩

Clears all items from the ⟨*comma list*⟩.

`\clist_clear_new:N`
`\clist_clear_new:c`
`\clist_gclear_new:N`
`\clist_gclear_new:c`

`\clist_clear_new:N` ⟨*comma list*⟩

Ensures that the ⟨*comma list*⟩ 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` ⟨comma list₁⟩ ⟨comma list₂⟩

Sets the content of ⟨comma list₁⟩ equal to that of ⟨comma list₂⟩.

`\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` ⟨comma list⟩ ⟨sequence⟩

Converts the data in the ⟨sequence⟩ into a ⟨comma list⟩: the original ⟨sequence⟩ 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` ⟨comma list₁⟩ ⟨comma list₂⟩ ⟨comma list₃⟩

Concatenates the content of ⟨comma list₂⟩ and ⟨comma list₃⟩ together and saves the result in ⟨comma list₁⟩. The items in ⟨comma list₂⟩ will be 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` ⟨comma list⟩
`\clist_if_exist:NTF` ⟨comma list⟩ {⟨true code⟩} {⟨false code⟩}

Tests whether the ⟨comma list⟩ is currently defined. This does not check that the ⟨comma list⟩ really is a comma list.

2 Adding data to comma lists

`\clist_set:Nn`
`\clist_set:(NV|No|Nx|cn|cV|co|cx)`
`\clist_gset:Nn`
`\clist_gset:(NV|No|Nx|cn|cV|co|cx)`
New: 2011-09-06

`\clist_set:Nn` ⟨comma list⟩ {⟨item₁⟩,...,⟨itemₙ⟩}

Sets ⟨comma list⟩ to contain the ⟨items⟩, removing any previous content from the variable. Spaces are removed from both sides of each item.

`\clist_put_left:Nn`
`\clist_put_left:(NV|No|Nx|cn|cV|co|cx)`
`\clist_gput_left:Nn`
`\clist_gput_left:(NV|No|Nx|cn|cV|co|cx)`
Updated: 2011-09-05

`\clist_put_left:Nn` ⟨comma list⟩ {⟨item₁⟩,...,⟨itemₙ⟩}

Appends the ⟨items⟩ to the left of the ⟨comma list⟩. Spaces are removed from both sides of each item.

`\clist_put_right:Nn`	`\clist_put_right:Nn` ⟨*comma list*⟩ `{`⟨*item₁*⟩`,...,`⟨*itemₙ*⟩`}`
`\clist_put_right:(NV|No|Nx|cn|cV|co|cx)`	
`\clist_gput_right:Nn`	
`\clist_gput_right:(NV|No|Nx|cn|cV|co|cx)`	

<div align="center">Updated: 2011-09-05</div>

Appends the ⟨*items*⟩ to the right of the ⟨*comma list*⟩. Spaces are removed from both sides of each item.

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:N` ⟨*comma list*⟩
`\clist_remove_duplicates:c`	
`\clist_gremove_duplicates:N`	
`\clist_gremove_duplicates:c`	

Removes duplicate items from the ⟨*comma list*⟩, leaving the left most copy of each item in the ⟨*comma list*⟩. The ⟨*item*⟩ comparison takes place on a token basis, as for `\tl_-if_eq:nn(TF)`.

TEXhackers note: This function iterates through every item in the ⟨*comma list*⟩ and does a comparison with the ⟨*items*⟩ already checked. It is therefore relatively slow with large comma lists. Furthermore, it will not work if any of the items in the ⟨*comma list*⟩ contains `{`, `}`, or `#` (assuming the usual TEX category codes apply).

`\clist_remove_all:Nn`	`\clist_remove_all:Nn` ⟨*comma list*⟩ `{`⟨*item*⟩`}`
`\clist_remove_all:cn`	
`\clist_gremove_all:Nn`	
`\clist_gremove_all:cn`	

<div align="center">Updated: 2011-09-06</div>

Removes every occurrence of ⟨*item*⟩ from the ⟨*comma list*⟩. The ⟨*item*⟩ comparison takes place on a token basis, as for `\tl_if_eq:nn(TF)`.

TEXhackers note: The ⟨*item*⟩ may not contain `{`, `}`, or `#` (assuming the usual TEX category codes apply).

`\clist_reverse:N`	`\clist_reverse:N` ⟨*comma list*⟩
`\clist_reverse:c`	
`\clist_greverse:N`	
`\clist_greverse:c`	

<div align="center">New: 2014-07-18</div>

Reverses the order of items stored in the ⟨*comma list*⟩.

`\clist_reverse:n`	`\clist_reverse:n` `{`⟨*comma list*⟩`}`

<div align="center">New: 2014-07-18</div>

Leaves the items in the ⟨*comma list*⟩ in the input stream in reverse order. Braces and spaces are preserved by this process.

TEXhackers note: The result is returned within `\unexpanded`, which means that the comma list will not expand further when appearing in an x-type argument expansion.

4 Comma list conditionals

`\clist_if_empty_p:N` ⋆
`\clist_if_empty_p:c` ⋆
`\clist_if_empty:NTF` ⋆
`\clist_if_empty:cTF` ⋆

`\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).

`\clist_if_empty_p:n` ⋆
`\clist_if_empty:nTF` ⋆

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:NnTF`
`\clist_if_in:(NV|No|cn|cV|co)TF`
`\clist_if_in:nnTF`
`\clist_if_in:(nV|no)TF`

Updated: 2011-09-06

`\clist_if_in:NnTF` ⟨*comma list*⟩ {⟨*item*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*item*⟩ is present in the ⟨*comma list*⟩. In the case of an n-type ⟨*comma list*⟩, spaces are stripped from each item, but braces are not removed. Hence,

> `\clist_if_in:nnTF { a , {b}~ , {b} , c } { b } {true} {false}`

yields `false`.

TEXhackers note: The ⟨*item*⟩ may not contain `{`, `}`, or `#` (assuming the usual TEX category codes apply), and should not contain `,` nor start or end with a space.

5 Mapping to comma lists

The functions described in this section apply a specified function to each item of a comma list.

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 your 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` ☆

Updated: 2012-06-29

`\clist_map_function:NN` ⟨*comma list*⟩ ⟨*function*⟩

Applies ⟨*function*⟩ to every ⟨*item*⟩ stored in the ⟨*comma list*⟩. The ⟨*function*⟩ will receive 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`. One mapping may be nested inside another.

`\clist_map_inline:Nn`
`\clist_map_inline:cn`
`\clist_map_inline:nn`

Updated: 2012-06-29

`\clist_map_inline:Nn` ⟨*comma list*⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*comma list*⟩. The ⟨*inline function*⟩ should consist of code which will receive the ⟨*item*⟩ as `#1`. One in line mapping can be nested inside another. 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` ⟨*comma list*⟩ ⟨*tl var.*⟩ {⟨*function using tl var.*⟩}

Stores each entry in the ⟨*comma list*⟩ in turn in the ⟨*tl var.*⟩ and applies the ⟨*function using tl var.*⟩ The ⟨*function*⟩ will usually consist of code making use of the ⟨*tl var.*⟩, but this is not enforced. One variable mapping can be nested inside another. The ⟨*items*⟩ are returned from left to right.

`\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 will normally take 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 will lead to low level TEX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\__prg_break_point:Nn` before further items are taken from the input stream. This will depend on the design of the mapping function.

`\clist_map_break:n` ☆

Updated: 2012-06-29

`\clist_map_break:n` {⟨*tokens*⟩}

Used to terminate a `\clist_map_`... function before all entries in the ⟨*comma list*⟩ have been processed, inserting the ⟨*tokens*⟩ after the mapping has ended. This will normally take place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \clist_map_break:n { <tokens> } }
      {
        % Do something useful
      }
  }
```

Use outside of a `\clist_map_`... scenario will lead to low level TEX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\__prg_break_point:Nn` before the ⟨*tokens*⟩ are inserted into the input stream. This will depend on the design of the mapping function.

`\clist_count:N` ⟨*comma list*⟩

Leaves the number of items in the ⟨*comma list*⟩ in the input stream as an ⟨*integer denotation*⟩. The total number of items in a ⟨*comma list*⟩ will include those which are duplicates, *i.e.* every item in a ⟨*comma list*⟩ is unique.

6 Using the content of comma lists directly

`\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 will be 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~ }
```

will insert "`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*⟩ will not expand further when appearing in an x-type argument expansion.

`\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 will be 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~ }
```

will insert "`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*⟩ will not expand further when appearing in an x-type argument expansion.

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

Updated: 2012-05-14

\clist_get:NN ⟨comma list⟩ ⟨token list variable⟩

Stores the left-most item from a ⟨comma list⟩ in the ⟨token list variable⟩ without removing it from the ⟨comma list⟩. The ⟨token list variable⟩ is assigned locally. If the ⟨comma list⟩ is empty the ⟨token list variable⟩ will contain the marker value \q_no_value.

\clist_get:NN*TF*
\clist_get:cN*TF*

New: 2012-05-14

\clist_get:NNTF ⟨comma list⟩ ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}

If the ⟨comma list⟩ is empty, leaves the ⟨false code⟩ in the input stream. The value of the ⟨token list variable⟩ is not defined in this case and should not be relied upon. If the ⟨comma list⟩ is non-empty, stores the top item from the ⟨comma list⟩ in the ⟨token list variable⟩ without removing it from the ⟨comma list⟩. The ⟨token list variable⟩ is assigned locally.

\clist_pop:NN
\clist_pop:cN

Updated: 2011-09-06

\clist_pop:NN ⟨comma list⟩ ⟨token list variable⟩

Pops the left-most item from a ⟨comma list⟩ into the ⟨token list variable⟩, *i.e.* removes the item from the comma list and stores it in the ⟨token list variable⟩. Both of the variables are assigned locally.

\clist_gpop:NN
\clist_gpop:cN

\clist_gpop:NN ⟨comma list⟩ ⟨token list variable⟩

Pops the left-most item from a ⟨comma list⟩ into the ⟨token list variable⟩, *i.e.* removes the item from the comma list and stores it in the ⟨token list variable⟩. The ⟨comma list⟩ is modified globally, while the assignment of the ⟨token list variable⟩ is local.

\clist_pop:NN*TF*
\clist_pop:cN*TF*

New: 2012-05-14

\clist_pop:NNTF ⟨comma list⟩ ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}

If the ⟨comma list⟩ is empty, leaves the ⟨false code⟩ in the input stream. The value of the ⟨token list variable⟩ is not defined in this case and should not be relied upon. If the ⟨comma list⟩ is non-empty, pops the top item from the ⟨comma list⟩ in the ⟨token list variable⟩, *i.e.* removes the item from the ⟨comma list⟩. Both the ⟨comma list⟩ and the ⟨token list variable⟩ are assigned locally.

\clist_gpop:NN*TF*
\clist_gpop:cN*TF*

New: 2012-05-14

\clist_gpop:NNTF ⟨comma list⟩ ⟨token list variable⟩ {⟨true code⟩} {⟨false code⟩}

If the ⟨comma list⟩ is empty, leaves the ⟨false code⟩ in the input stream. The value of the ⟨token list variable⟩ is not defined in this case and should not be relied upon. If the ⟨comma list⟩ is non-empty, pops the top item from the ⟨comma list⟩ in the ⟨token list variable⟩, *i.e.* removes the item from the ⟨comma list⟩. The ⟨comma list⟩ is modified globally, while the ⟨token list variable⟩ is assigned locally.

`\clist_push:Nn`
`\clist_push:(NV|No|Nx|cn|cV|co|cx)`
`\clist_gpush:Nn`
`\clist_gpush:(NV|No|Nx|cn|cV|co|cx)`

`\clist_push:Nn ⟨comma list⟩ {⟨items⟩}`

Adds the {⟨*items*⟩} to the top of the ⟨*comma list*⟩. Spaces are removed from both sides of each item.

8 Using a single item

`\clist_item:Nn ⋆`
`\clist_item:cn ⋆`
`\clist_item:nn ⋆`

New: 2014-07-17

`\clist_item:Nn ⟨comma list⟩ {⟨integer expression⟩}`

Indexing items in the ⟨*comma list*⟩ from 1 at the top (left), this function will evaluate the ⟨*integer expression*⟩ and leave the appropriate item from the comma list in the input stream. If the ⟨*integer expression*⟩ is negative, indexing occurs from the bottom (right) of the comma list. When the ⟨*integer expression*⟩ is larger than the number of items in the ⟨*comma list*⟩ (as calculated by `\clist_count:N`) then the function will expand to nothing.

TEXhackers note: The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*item*⟩ will not expand further when appearing in an x-type argument expansion.

9 Viewing comma lists

`\clist_show:N`
`\clist_show:c`

Updated: 2015-08-03

`\clist_show:N ⟨comma list⟩`

Displays the entries in the ⟨*comma list*⟩ in the terminal.

`\clist_show:n`

Updated: 2013-08-03

`\clist_show:n {⟨tokens⟩}`

Displays the entries in the comma list in the terminal.

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.

Part XV

The **l3prop** package
Property lists

LaTeX3 implements a "property list" data type, which contain an unordered list of entries each of which consists of a ⟨*key*⟩ and an associated ⟨*value*⟩. The ⟨*key*⟩ and ⟨*value*⟩ may both be any ⟨*balanced text*⟩. It is possible to map functions to property lists such that the function is applied 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 will overwrite the existing one. The ⟨*keys*⟩ are compared on a string basis, using the same method as `\str_if_eq:nn`.

Property lists are intended for storing key-based information for use within code. This is in contrast to key–value lists, which are a form of *input* parsed by the keys module.

1 Creating and initialising property lists

`\prop_new:N`
`\prop_new:c`

`\prop_new:N` ⟨*property list*⟩

Creates a new ⟨*property list*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*property list*⟩ will initially contain no 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.

`\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 list₁*⟩ ⟨*property list₂*⟩

Sets the content of ⟨*property list₁*⟩ equal to that of ⟨*property list₂*⟩.

2 Adding entries to property lists

`\prop_put:Nnn`
`\prop_put:(NnV|Nno|Nnx|NVn|NVV|Non|Noo|cnn|cnV|cno|cnx|cVn|cVV|con|coo)`
`\prop_gput:Nnn`
`\prop_gput:(NnV|Nno|Nnx|NVn|NVV|Non|Noo|cnn|cnV|cno|cnx|cVn|cVV|con|coo)`

`\prop_put:Nnn` ⟨property list⟩
`{`⟨key⟩`}` `{`⟨value⟩`}`

Updated: 2012-07-09

Adds an entry to the ⟨*property list*⟩ which may be accessed using the ⟨*key*⟩ and which has ⟨*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. If the ⟨*key*⟩ is already present in the ⟨*property list*⟩, the existing entry is overwritten by the new ⟨*value*⟩.

`\prop_put_if_new:Nnn`
`\prop_put_if_new:cnn`
`\prop_gput_if_new:Nnn`
`\prop_gput_if_new:cnn`

`\prop_put_if_new:Nnn` ⟨property list⟩ `{`⟨key⟩`}` `{`⟨value⟩`}`

If the ⟨*key*⟩ is present in the ⟨*property list*⟩ then no action is taken. If the ⟨*key*⟩ is not present in the ⟨*property list*⟩ then a new entry is added. 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.

3 Recovering values from property lists

`\prop_get:NnN`
`\prop_get:(NVN|NoN|cnN|cVN|coN)`

`\prop_get:NnN` ⟨property list⟩ `{`⟨key⟩`}` ⟨tl var⟩

Updated: 2011-08-28

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 ⟨*token list variable*⟩ will contain 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:(NoN|cnN|coN)`

`\prop_pop:NnN` ⟨property list⟩ `{`⟨key⟩`}` ⟨tl var⟩

Updated: 2011-08-18

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 ⟨*token list variable*⟩ will contain 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:(NoN|cnN|coN)`

`\prop_gpop:NnN` ⟨property list⟩ `{`⟨key⟩`}` ⟨tl var⟩

Updated: 2011-08-18

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 ⟨*token list variable*⟩ will contain 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 ⟨*token list variable*⟩ is local. See also `\prop_gpop:NnNTF`.

`\prop_item:Nn` ⋆	
`\prop_item:cn` ⋆	
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: This function is slower than the non-expandable analogue `\prop_-get:NnN`. The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨*value*⟩ will not expand further when appearing in an x-type argument expansion.

4   Modifying property lists

`\prop_remove:Nn`
`\prop_remove:(NV|cn|cV)`
`\prop_gremove:Nn`
`\prop_gremove:(NV|cn|cV)`
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.

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|No|cn|cV|co)`	⋆
`\prop_if_in:NnTF`	⋆
`\prop_if_in:(NV|No|cn|cV|co)TF`	⋆
Updated: 2011-09-15	

`\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: This function iterates through every key–value pair in the ⟨*property list*⟩ and is therefore slower than using the non-expandable `\prop_get:NnNTF`.

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 cases follow dependent 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:NnN*TF*
\prop_get:(NVN|NoN|cnN|cVN|coN)*TF*

Updated: 2012-05-19

\prop_get:NnNTF ⟨property list⟩ {⟨key⟩} ⟨token list variable⟩
  {⟨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 ⟨token list variable⟩ 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 ⟨token list variable⟩ without removing it from the ⟨property list⟩, then leaves the ⟨true code⟩ in the input stream. The ⟨token list variable⟩ is assigned locally.

\prop_pop:NnN*TF*
\prop_pop:cnN*TF*

New: 2011-08-18
Updated: 2012-05-19

\prop_pop:NnNTF ⟨property list⟩ {⟨key⟩} ⟨token list variable⟩ {⟨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 ⟨token list variable⟩ 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 ⟨token list variable⟩, *i.e.* removes the item from the ⟨property list⟩. Both the ⟨property list⟩ and the ⟨token list variable⟩ are assigned locally.

\prop_gpop:NnN*TF*
\prop_gpop:cnN*TF*

New: 2011-08-18
Updated: 2012-05-19

\prop_gpop:NnNTF ⟨property list⟩ {⟨key⟩} ⟨token list variable⟩ {⟨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 ⟨token list variable⟩ 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 ⟨token list variable⟩, *i.e.* removes the item from the ⟨property list⟩. The ⟨property list⟩ is modified globally, while the ⟨token list variable⟩ is assigned locally.

7 Mapping to property lists

\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⟩ will receive two argument for each iteration: the ⟨key⟩ and associated ⟨value⟩. The order in which ⟨entries⟩ are returned is not defined and should not be relied upon.

\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 will receive 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_break: ☆

Updated: 2012-06-29

`\prop_map_break:`

Used to terminate a `\prop_map_...` function before all entries in the ⟨*property list*⟩ have been processed. This will normally take 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 will lead to low level TEX errors.

\prop_map_break:n ☆

Updated: 2012-06-29

`\prop_map_break:n {⟨tokens⟩}`

Used to terminate a `\prop_map_...` function before all entries in the ⟨*property list*⟩ have been processed, inserting the ⟨*tokens*⟩ after the mapping has ended. This will normally take place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \prop_map_break:n { <tokens> } }
      {
        % Do something useful
      }
  }
```

Use outside of a `\prop_map_...` scenario will lead to low level TEX errors.

8 Viewing property lists

\prop_show:N
\prop_show:c

Updated: 2015-08-01

`\prop_show:N ⟨property list⟩`

Displays the entries in the ⟨*property list*⟩ in the terminal.

9 Scratch property lists

\l_tmpa_prop
\l_tmpb_prop

New: 2012-06-23

Scratch 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 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.

10   Constants

\c_empty_prop

A permanently-empty property list used for internal comparisons.

11   Internal property list functions

\s__prop

The internal token used at the beginning of property lists. This is also used after each ⟨*key*⟩ (see __prop_pair:wn).

__prop_pair:wn

__prop_pair:wn ⟨key⟩ \s__prop {⟨item⟩}

The internal token used to begin each key–value pair in the property list. If expanded outside of a mapping or manipulation function, an error will be raised. The definition should always be set globally.

\l__prop_internal_tl

Token list used to store new key–value pairs to be inserted by functions of the \prop_-put:Nnn family.

__prop_split:NnTF

Updated: 2013-01-08

__prop_split:NnTF ⟨property list⟩ {⟨key⟩} {⟨true code⟩} {⟨false code⟩}

Splits the ⟨*property list*⟩ at the ⟨*key*⟩, giving three token lists: the ⟨*extract*⟩ of ⟨*property list*⟩ before the ⟨*key*⟩, the ⟨*value*⟩ associated with the ⟨*key*⟩ and the ⟨*extract*⟩ of the ⟨*property list*⟩ after the ⟨*value*⟩. Both ⟨*extracts*⟩ retain the internal structure of a property list, and the concatenation of the two ⟨*extracts*⟩ is a property list. 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 first ⟨*extract*⟩, the ⟨*value*⟩, and the second extract. If the ⟨*key*⟩ is not present in the ⟨*property list*⟩ then the ⟨*false code*⟩ is left in the input stream, with no trailing material. Both ⟨*true code*⟩ and ⟨*false code*⟩ are used in the replacement text of a macro defined internally, hence macro parameter characters should be doubled, except #1, #2, and #3 which stand in the ⟨*true code*⟩ for the three extracts from the property list. The ⟨*key*⟩ comparison takes place as described for \str_if_eq:nn.

Part XVI

The **l3box** package
Boxes

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_`.

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⟩ will initially be 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_void_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 ⟨box₁⟩ ⟨box₂⟩`

Sets the content of ⟨box₁⟩ equal to that of ⟨box₂⟩.

`\box_set_eq_clear:NN`
`\box_set_eq_clear:(cN|Nc|cc)`

`\box_set_eq_clear:NN ⟨box₁⟩ ⟨box₂⟩`

Sets the content of ⟨box₁⟩ within the current TeX group equal to that of ⟨box₂⟩, then clears ⟨box₂⟩ globally.

`\box_gset_eq_clear:NN`
`\box_gset_eq_clear:(cN|Nc|cc)`

`\box_gset_eq_clear:NN ⟨box₁⟩ ⟨box₂⟩`

Sets the content of ⟨box₁⟩ equal to that of ⟨box₂⟩, then clears ⟨box₂⟩. These assignments are global.

`\box_if_exist_p:N ⋆`
`\box_if_exist_p:c ⋆`
`\box_if_exist:NTF ⋆`
`\box_if_exist:cTF ⋆`

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.

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.

TₑXhackers note: This is the TₑX primitive \copy.

\box_use_clear:N
\box_use_clear:c

\box_use_clear:N ⟨box⟩

Inserts the current content of the ⟨box⟩ onto the current list for typesetting, then globally clears the content of the ⟨box⟩.

TₑXhackers note: This is the TₑX primitive \box.

\box_move_right:nn
\box_move_left:nn

\box_move_right:nn {⟨dimexpr⟩} {⟨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 ⟨dimexpr⟩ 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 {⟨dimexpr⟩} {⟨box function⟩}

This function operates in horizontal mode, and inserts the material specified by the ⟨box function⟩ such that its reference point is displaced vertical by the given ⟨dimexpr⟩ 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 }.

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 ⟨dimension expression⟩.

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 ⟨dimension expression⟩.

TₑXhackers note: This is the TₑX primitive \ht.

`\box_wd:N`   `\box_wd:N ⟨box⟩`
`\box_wd:c`

Calculates the width of the ⟨box⟩ in a form suitable for use in a ⟨*dimension expression*⟩.

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

`\box_set_dp:Nn`   `\box_set_dp:Nn ⟨box⟩ {⟨dimension expression⟩}`
`\box_set_dp:cn`

Updated: 2011-10-22

Set the depth (below the baseline) of the ⟨box⟩ to the value of the {⟨*dimension expression*⟩}. This is a global assignment.

`\box_set_ht:Nn`   `\box_set_ht:Nn ⟨box⟩ {⟨dimension expression⟩}`
`\box_set_ht:cn`

Updated: 2011-10-22

Set the height (above the baseline) of the ⟨box⟩ to the value of the {⟨*dimension expression*⟩}. This is a global assignment.

`\box_set_wd:Nn`   `\box_set_wd:Nn ⟨box⟩ {⟨dimension expression⟩}`
`\box_set_wd:cn`

Updated: 2011-10-22

Set the width of the ⟨box⟩ to the value of the {⟨*dimension expression*⟩}. This is a global assignment.

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.

5 The last box inserted

`\box_set_to_last:N`   `\box_set_to_last:N ⟨box⟩`
`\box_set_to_last:c`
`\box_gset_to_last:N`
`\box_gset_to_last:c`

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⟩ will always be void as it is not possible to recover the last added item.

6 Constant boxes

\c_empty_box

Updated: 2012-11-04

This is a permanently empty box, which is neither set as horizontal nor vertical.

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.

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⟩ ⟨intexpr₁⟩ ⟨intexpr₂⟩

Display the contents of ⟨box⟩ in the terminal, showing the first ⟨intexpr₁⟩ items of the box, and descending into ⟨intexpr₂⟩ 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⟩ ⟨intexpr₁⟩ ⟨intexpr₂⟩

Writes the contents of ⟨box⟩ to the log, showing the first ⟨intexpr₁⟩ items of the box, and descending into ⟨intexpr₂⟩ group levels.

9 Horizontal mode boxes

\hbox:n

\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`	`\hbox_to_wd:nn {⟨dimexpr⟩} {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a horizontal box of width ⟨*dimexpr*⟩ and then includes this box in the current list for typesetting.

`\hbox_to_zero:n`	`\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:Nn ⟨box⟩ {⟨contents⟩}`
`\hbox_set:cn`	
`\hbox_gset:Nn`	Typesets the ⟨*contents*⟩ at natural width and then stores the result inside the ⟨*box*⟩.
`\hbox_gset:cn`	

`\hbox_set_to_wd:Nnn`	`\hbox_set_to_wd:Nnn ⟨box⟩ {⟨dimexpr⟩} {⟨contents⟩}`
`\hbox_set_to_wd:cnn`	
`\hbox_gset_to_wd:Nnn`	Typesets the ⟨*contents*⟩ to the width given by the ⟨*dimexpr*⟩ and then stores the result
`\hbox_gset_to_wd:cnn`	inside the ⟨*box*⟩.

`\hbox_overlap_right:n`	`\hbox_overlap_right:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a horizontal box of zero width such that material will protrude to the right of the insertion point.

`\hbox_overlap_left:n`	`\hbox_overlap_left:n {⟨contents⟩}`

Typesets the ⟨*contents*⟩ into a horizontal box of zero width such that material will protrude to the left of the insertion point.

`\hbox_set:Nw`	`\hbox_set:Nw ⟨box⟩ ⟨contents⟩ \hbox_set_end:`
`\hbox_set:cw`	
`\hbox_set_end:`	Typesets the ⟨*contents*⟩ at natural width and then stores the result inside the ⟨*box*⟩. In
`\hbox_gset:Nw`	contrast to `\hbox_set:Nn` this function does not absorb the argument when finding the
`\hbox_gset:cw`	⟨*content*⟩, and so can be used in circumstances where the ⟨*content*⟩ may not be a simple
`\hbox_gset_end:`	argument.

`\hbox_unpack:N`	`\hbox_unpack:N ⟨box⟩`
`\hbox_unpack:c`	

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`.

`\hbox_unpack_clear:N`	`\hbox_unpack_clear:N ⟨box⟩`
`\hbox_unpack_clear:c`	

Unpacks the content of the horizontal ⟨*box*⟩, retaining any stretching or shrinking applied when the ⟨*box*⟩ was set. The ⟨*box*⟩ is then cleared globally.

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

10 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 will have 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 will typically be non-zero.

\vbox:n

Updated: 2011-12-18

\vbox:n {⟨contents⟩}

Typesets the ⟨contents⟩ into a vertical box of natural height and includes this box in the current list for typesetting.

TEXhackers note: This is the TEX primitive \vbox.

\vbox_top:n

Updated: 2011-12-18

\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 will be equal to that of the *first* item added to the box.

TEXhackers note: This is the TEX primitive \vtop.

\vbox_to_ht:nn

Updated: 2011-12-18

\vbox_to_ht:nn {⟨dimexpr⟩} {⟨contents⟩}

Typesets the ⟨contents⟩ into a vertical box of height ⟨dimexpr⟩ and then includes this box in the current list for typesetting.

\vbox_to_zero:n

Updated: 2011-12-18

\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: 2011-12-18

\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: 2011-12-18

\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 will be 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: 2011-12-18

\vbox_set_to_ht:Nnn ⟨box⟩ {⟨dimexpr⟩} {⟨contents⟩}

Typesets the ⟨contents⟩ to the height given by the ⟨dimexpr⟩ 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: 2011-12-18

\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_split_to_ht:NNn

Updated: 2011-10-22

\vbox_set_split_to_ht:NNn ⟨box₁⟩ ⟨box₂⟩ {⟨dimexpr⟩}

Sets ⟨box₁⟩ to contain material to the height given by the ⟨dimexpr⟩ by removing content from the top of ⟨box₂⟩ (which must be a vertical box).

TEXhackers note: This is the TEX primitive \vsplit.

\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.

\vbox_unpack_clear:N
\vbox_unpack_clear:c

\vbox_unpack:N ⟨box⟩

Unpacks the content of the vertical ⟨box⟩, retaining any stretching or shrinking applied when the ⟨box⟩ was set. The ⟨box⟩ is then cleared globally.

TEXhackers note: This is the TEX primitive \unvbox.

11   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`.

Part XVII

The **l3coffins** package
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).

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⟩ will initially be empty.

\coffin_clear:N
\coffin_clear:c

New: 2011-08-17

\coffin_clear:N ⟨coffin⟩

Clears the content of the ⟨coffin⟩ within the current TEX group level.

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

New: 2011-08-17

\coffin_set_eq:NN ⟨coffin₁⟩ ⟨coffin₂⟩

Sets both the content and poles of ⟨coffin₁⟩ equal to those of ⟨coffin₂⟩ within the current TEX group level.

\coffin_if_exist_p:N ⋆
\coffin_if_exist_p:c ⋆
\coffin_if_exist:N*TF* ⋆
\coffin_if_exist:c*TF* ⋆

New: 2012-06-20

\coffin_if_exist_p:N ⟨box⟩
\coffin_if_exist:NTF ⟨box⟩ {⟨true code⟩} {⟨false code⟩}

Tests whether the ⟨coffin⟩ is currently defined.

2 Setting coffin content and poles

All coffin functions create and manipulate coffins locally within the current TEX group level.

\hcoffin_set:Nn
\hcoffin_set:cn

New: 2011-08-17
Updated: 2011-09-03

\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:

New: 2011-09-10

\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

New: 2011-08-17
Updated: 2012-05-22

`\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:

New: 2011-09-10
Updated: 2012-05-22

`\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

New: 2012-07-20

`\coffin_set_horizontal_pole:Nnn ⟨coffin⟩`
`    {⟨pole⟩} {⟨offset⟩}`

Sets the ⟨*pole*⟩ to run horizontally through the ⟨*coffin*⟩. The ⟨*pole*⟩ will be located at the ⟨*offset*⟩ from the bottom edge of the bounding box of the ⟨*coffin*⟩. The ⟨*offset*⟩ should be given as a dimension expression.

\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnn

New: 2012-07-20

`\coffin_set_vertical_pole:Nnn ⟨coffin⟩ {⟨pole⟩} {⟨offset⟩}`

Sets the ⟨*pole*⟩ to run vertically through the ⟨*coffin*⟩. The ⟨*pole*⟩ will be located at the ⟨*offset*⟩ from the left-hand edge of the bounding box of the ⟨*coffin*⟩. The ⟨*offset*⟩ should be given as a dimension expression.

3 Joining and using coffins

\coffin_attach:NnnNnnnn
\coffin_attach:(cnnNnnnn|Nnncnnnn|cnncnnnn)

`\coffin_attach:NnnNnnnn`
`    ⟨coffin₁⟩ {⟨coffin₁-pole₁⟩} {⟨coffin₁-pole₂⟩}`
`    ⟨coffin₂⟩ {⟨coffin₂-pole₁⟩} {⟨coffin₂-pole₂⟩}`
`    {⟨x-offset⟩} {⟨y-offset⟩}`

This function attaches ⟨*coffin₂*⟩ to ⟨*coffin₁*⟩ such that the bounding box of ⟨*coffin₁*⟩ is not altered, *i.e.* ⟨*coffin₂*⟩ can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating ⟨*handle₁*⟩, the point of intersection of ⟨*coffin₁-pole₁*⟩ and ⟨*coffin₁-pole₂*⟩, and ⟨*handle₂*⟩, the point of intersection of ⟨*coffin₂-pole₁*⟩ and ⟨*coffin₂-pole₂*⟩. ⟨*coffin₂*⟩ is then attached to ⟨*coffin₁*⟩ such that the relationship between ⟨*handle₁*⟩ and ⟨*handle₂*⟩ 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_join:NnnNnnnn`
$\langle coffin_1\rangle$ `{`$\langle coffin_1$-$pole_1\rangle$`}` `{`$\langle coffin_1$-$pole_2\rangle$`}`
$\langle coffin_2\rangle$ `{`$\langle coffin_2$-$pole_1\rangle$`}` `{`$\langle coffin_2$-$pole_2\rangle$`}`
`{`$\langle x\text{-}offset\rangle$`}` `{`$\langle y\text{-}offset\rangle$`}`

This function joins $\langle coffin_2\rangle$ to $\langle coffin_1\rangle$ such that the bounding box of $\langle coffin_1\rangle$ may expand. The new bounding box will cover the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating $\langle handle_1\rangle$, the point of intersection of $\langle coffin_1$-$pole_1\rangle$ and $\langle coffin_1$-$pole_2\rangle$, and $\langle handle_2\rangle$, the point of intersection of $\langle coffin_2$-$pole_1\rangle$ and $\langle coffin_2$-$pole_2\rangle$. $\langle coffin_2\rangle$ is then attached to $\langle coffin_1\rangle$ such that the relationship between $\langle handle_1\rangle$ and $\langle handle_2\rangle$ is described by the $\langle x\text{-}offset\rangle$ and $\langle y\text{-}offset\rangle$. The two offsets should be given as dimension expressions.

`\coffin_typeset:Nnnnn`
`\coffin_typeset:cnnnn`

Updated: 2012-07-20

`\coffin_typeset:Nnnnn` $\langle coffin\rangle$ `{`$\langle pole_1\rangle$`}` `{`$\langle pole_2\rangle$`}`
`{`$\langle x\text{-}offset\rangle$`}` `{`$\langle y\text{-}offset\rangle$`}`

Typesetting is carried out by first calculating $\langle handle\rangle$, the point of intersection of $\langle pole_1\rangle$ and $\langle pole_2\rangle$. The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the $\langle handle\rangle$ is described by the $\langle x\text{-}offset\rangle$ and $\langle y\text{-}offset\rangle$. 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.

4 Measuring coffins

`\coffin_dp:N`
`\coffin_dp:c`

`\coffin_dp:N` $\langle coffin\rangle$

Calculates the depth (below the baseline) of the $\langle coffin\rangle$ in a form suitable for use in a $\langle dimension\ expression\rangle$.

`\coffin_ht:N`
`\coffin_ht:c`

`\coffin_ht:N` $\langle coffin\rangle$

Calculates the height (above the baseline) of the $\langle coffin\rangle$ in a form suitable for use in a $\langle dimension\ expression\rangle$.

`\coffin_wd:N`
`\coffin_wd:c`

`\coffin_wd:N` $\langle coffin\rangle$

Calculates the width of the $\langle coffin\rangle$ in a form suitable for use in a $\langle dimension\ expression\rangle$.

5 Coffin diagnostics

`\coffin_display_handles:Nn`
`\coffin_display_handles:cn`

Updated: 2011-09-02

`\coffin_display_handles:Nn` $\langle coffin\rangle$ `{`$\langle color\rangle$`}`

This function first calculates the intersections between all of the $\langle poles\rangle$ of the $\langle coffin\rangle$ to give a set of $\langle handles\rangle$. It then prints the $\langle coffin\rangle$ at the current location in the source, with the position of the $\langle handles\rangle$ marked on the coffin. The $\langle handles\rangle$ will be labelled as part of this process: the locations of the $\langle handles\rangle$ and the labels are both printed in the $\langle color\rangle$ specified.

\coffin_mark_handle:Nnnn
\coffin_mark_handle:cnnn

Updated: 2011-09-02

\coffin_mark_handle:Nnnn ⟨*coffin*⟩ {⟨*pole₁*⟩} {⟨*pole₂*⟩} {⟨*color*⟩}

This function first calculates the ⟨*handle*⟩ for the ⟨*coffin*⟩ as defined by the intersection of ⟨*pole₁*⟩ and ⟨*pole₂*⟩. It then marks the position of the ⟨*handle*⟩ on the ⟨*coffin*⟩. The ⟨*handle*⟩ will be 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 co-ordinates 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.

5.1 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.

Part XVIII

The **l3color** package
Color support

This module provides support for color in LaTeX3. At present, the material here is mainly intended to support a small number of low-level requirements in other l3kernel modules.

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.

\color_ensure_current:

New: 2011-09-03

```
\color_ensure_current:
```

Ensures that material inside a box will use 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.

Part XIX
The **l3msg** package 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.

1 Creating new messages

All messages have to be created before they can be used. The text of messages will automatically by wrapped to the length available in the console. As a result, formatting is only needed where it will help 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 only those messages from the `submodule` to be filtered out.

`\msg_new:nnnn`
`\msg_new:nnn`

Updated: 2011-08-16

`\msg_new:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Creates a ⟨*message*⟩ for a given ⟨*module*⟩. The message will be 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 will be raised if the ⟨*message*⟩ already exists.

`\msg_set:nnnn`
`\msg_set:nnn`
`\msg_gset:nnnn`
`\msg_gset:nnn`

`\msg_set:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Sets up the text for a ⟨*message*⟩ for a given ⟨*module*⟩. The message will be 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_p:nn {⟨module⟩} {⟨message⟩}`
`\msg_if_exist:nnTF` ⋆	`\msg_if_exist:nnTF {⟨module⟩} {⟨message⟩} {⟨true code⟩} {⟨false code⟩}`
New: 2012-03-03	Tests whether the ⟨*message*⟩ for the ⟨*module*⟩ is currently defined.

2 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 ⟨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.

`\msg_critical_text:n` ⋆   `\msg_critical_text:n {⟨module⟩}`

Produces the standard text

> `Critical ⟨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.

`\msg_error_text:n` ⋆   `\msg_error_text:n {⟨module⟩}`

Produces the standard text

> `⟨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.

`\msg_warning_text:n` ⋆   `\msg_warning_text:n {⟨module⟩}`

Produces the standard text

> `⟨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.

\msg_info_text:n ⋆   \msg_info_text:n {⟨module⟩}

Produces the standard text:

⟨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.

\msg_see_documentation_text:n ⋆   \msg_see_documentation_text:n {⟨module⟩}

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.

3 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 will be ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments will be converted to strings before being added to the message text: the x-type variants should be used to expand material.

\msg_fatal:nnnnnn
\msg_fatal:nnxxxx
\msg_fatal:nnnnn
\msg_fatal:nnxxx
\msg_fatal:nnnn
\msg_fatal:nnxx
\msg_fatal:nnn
\msg_fatal:nnx
\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 will halt.

\msg_critical:nnnnnn
\msg_critical:nnxxxx
\msg_critical:nnnnn
\msg_critical:nnxxx
\msg_critical:nnnn
\msg_critical:nnxx
\msg_critical:nnn
\msg_critical:nnx
\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 will stop 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:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩}`
`\msg_error:nnxxxx`	`{⟨arg four⟩}`
`\msg_error:nnnnn`	Issues ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating
`\msg_error:nnxxx`	functions. The error will interrupt processing and issue the text at the terminal. After
`\msg_error:nnnn`	user input, the run will continue.
`\msg_error:nnxx`	
`\msg_error:nnn`	
`\msg_error:nnx`	
`\msg_error:nn`	

Updated: 2012-08-11

`\msg_warning:nnnnnn`	`\msg_warning:nnxxxx {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩}`
`\msg_warning:nnxxxx`	`{⟨arg four⟩}`
`\msg_warning:nnnnn`	Issues ⟨*module*⟩ warning ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating
`\msg_warning:nnxxx`	functions. The warning text will be added to the log file and the terminal, but the TeX
`\msg_warning:nnnn`	run will not be interrupted.
`\msg_warning:nnxx`	
`\msg_warning:nnn`	
`\msg_warning:nnx`	
`\msg_warning:nn`	

Updated: 2012-08-11

`\msg_info:nnnnnn`	`\msg_info:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg`
`\msg_info:nnxxxx`	`four⟩}`
`\msg_info:nnnnn`	Issues ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating
`\msg_info:nnxxx`	functions. The information text will be added to the log file.
`\msg_info:nnnn`	
`\msg_info:nnxx`	
`\msg_info:nnn`	
`\msg_info:nnx`	
`\msg_info:nn`	

Updated: 2012-08-11

`\msg_log:nnnnnn`	`\msg_log:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg`
`\msg_log:nnxxxx`	`four⟩}`
`\msg_log:nnnnn`	Issues ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating
`\msg_log:nnxxx`	functions. The information text will be added to the log file: the output is briefer than
`\msg_log:nnnn`	`\msg_info:nnnnnn`.
`\msg_log:nnxx`	
`\msg_log:nnn`	
`\msg_log:nnx`	
`\msg_log: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).

4 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 will raise 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 will raise errors 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 {⟨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⟩.

154

\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 }
```

5 Low-level message functions

The lower-level message functions should usually be accessed from the higher-level system. However, there are occasions where direct access to these functions is desirable.

\msg_interrupt:nnn

New: 2012-06-28

`\msg_interrupt:nnn {⟨first line⟩} {⟨text⟩} {⟨extra text⟩}`

Interrupts the TeX run, issuing a formatted message comprising ⟨*first line*⟩ and ⟨*text*⟩ laid out in the format

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! <first line>
!
! <text>
!.........................................
```

where the ⟨*text*⟩ will be wrapped to fit within the current line length. The user may then request more information, at which stage the ⟨*extra text*⟩ will be shown in the terminal in the format

```
|,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
|  <extra text>
|.........................................
```

where the ⟨*extra text*⟩ will be wrapped within the current line length. Wrapping of both ⟨*text*⟩ and ⟨*more text*⟩ takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

\msg_log:n

New: 2012-06-28

`\msg_log:n {⟨text⟩}`

Writes to the log file with the ⟨*text*⟩ laid out in the format

```
.................................................
. <text>
.................................................
```

where the ⟨*text*⟩ will be wrapped to fit within the current line length. Wrapping takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

\msg_term:n

New: 2012-06-28

`\msg_term:n {⟨text⟩}`

Writes to the terminal and log file with the ⟨*text*⟩ laid out in the format

```
*************************************************
* <text>
*************************************************
```

where the ⟨*text*⟩ will be wrapped to fit within the current line length. Wrapping takes place using `\iow_wrap:nnnN`; the documentation for the latter should be consulted for full details.

6 Kernel-specific functions

Messages from LaTeX3 itself are handled by the general message system, but have their own functions. This allows some text to be pre-defined, and also ensures that serious errors can be handled properly.

__msg_kernel_new:nnnn
__msg_kernel_new:nnn

Updated: 2011-08-16

`\__msg_kernel_new:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Creates a kernel ⟨*message*⟩ for a given ⟨*module*⟩. The message will be 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 and expanded at the time the message is used. An error will be raised if the ⟨*message*⟩ already exists.

__msg_kernel_set:nnnn
__msg_kernel_set:nnn

`\__msg_kernel_set:nnnn {⟨module⟩} {⟨message⟩} {⟨text⟩} {⟨more text⟩}`

Sets up the text for a kernel ⟨*message*⟩ for a given ⟨*module*⟩. The message will be 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 and expanded at the time the message is used.

`\__msg_kernel_fatal:nnnnnn`	`\__msg_kernel_fatal:nnnnnn` `{⟨module⟩}` `{⟨message⟩}` `{⟨arg one⟩}` `{⟨arg two⟩}` `{⟨arg`
`\__msg_kernel_fatal:nnxxxx`	`three⟩}` `{⟨arg four⟩}`
`\__msg_kernel_fatal:nnnnn`	
`\__msg_kernel_fatal:nnxxx`	Issues kernel ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating
`\__msg_kernel_fatal:nnnn`	functions. After issuing a fatal error the TEX run will halt. Cannot be redirected.
`\__msg_kernel_fatal:nnxx`	
`\__msg_kernel_fatal:nnn`	
`\__msg_kernel_fatal:nnx`	
`\__msg_kernel_fatal:nn`	

Updated: 2012-08-11

`\__msg_kernel_error:nnnnnn`	`\__msg_kernel_error:nnnnnn` `{⟨module⟩}` `{⟨message⟩}` `{⟨arg one⟩}` `{⟨arg two⟩}` `{⟨arg`
`\__msg_kernel_error:nnxxxx`	`three⟩}` `{⟨arg four⟩}`
`\__msg_kernel_error:nnnnn`	
`\__msg_kernel_error:nnxxx`	Issues kernel ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating
`\__msg_kernel_error:nnnn`	functions. The error will stop processing and issue the text at the terminal. After user
`\__msg_kernel_error:nnxx`	input, the run will continue. Cannot be redirected.
`\__msg_kernel_error:nnn`	
`\__msg_kernel_error:nnx`	
`\__msg_kernel_error:nn`	

Updated: 2012-08-11

`\__msg_kernel_warning:nnnnnn`	`\__msg_kernel_warning:nnnnnn` `{⟨module⟩}` `{⟨message⟩}` `{⟨arg one⟩}` `{⟨arg`
`\__msg_kernel_warning:nnxxxx`	`two⟩}` `{⟨arg three⟩}` `{⟨arg four⟩}`
`\__msg_kernel_warning:nnnnn`	
`\__msg_kernel_warning:nnxxx`	
`\__msg_kernel_warning:nnnn`	
`\__msg_kernel_warning:nnxx`	
`\__msg_kernel_warning:nnn`	
`\__msg_kernel_warning:nnx`	
`\__msg_kernel_warning:nn`	

Updated: 2012-08-11

Issues kernel ⟨*module*⟩ warning ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The warning text will be added to the log file, but the TEX run will not be interrupted.

`\__msg_kernel_info:nnnnnn`	`\__msg_kernel_info:nnnnnn` `{⟨module⟩}` `{⟨message⟩}` `{⟨arg one⟩}` `{⟨arg two⟩}` `{⟨arg`
`\__msg_kernel_info:nnxxxx`	`three⟩}` `{⟨arg four⟩}`
`\__msg_kernel_info:nnnnn`	
`\__msg_kernel_info:nnxxx`	Issues kernel ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the
`\__msg_kernel_info:nnnn`	text-creating functions. The information text will be added to the log file.
`\__msg_kernel_info:nnxx`	
`\__msg_kernel_info:nnn`	
`\__msg_kernel_info:nnx`	
`\__msg_kernel_info:nn`	

Updated: 2012-08-11

157

7 Expandable errors

In a few places, the LaTeX3 kernel needs to produce errors in an expansion only context. This 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. However, the interface is similar, with the important caveat that the message text and arguments are not expanded, and messages should be very short.

__msg_kernel_expandable_error:nnnnnn ⋆
__msg_kernel_expandable_error:nnnnn ⋆
__msg_kernel_expandable_error:nnnn ⋆
__msg_kernel_expandable_error:nnn ⋆
__msg_kernel_expandable_error:nn ⋆

New: 2011-11-23

__msg_kernel_expandable_error:nnnnnn {⟨module⟩} {⟨message⟩}
{⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}

Issues an error, passing ⟨arg one⟩ to ⟨arg four⟩ to the text-creating functions. The resulting string must be shorter than a line, otherwise it will be cropped.

__msg_expandable_error:n ⋆

New: 2011-08-11
Updated: 2011-08-13

__msg_expandable_error:n {⟨error message⟩}

Issues an "Undefined error" message from TeX itself, and prints the ⟨error message⟩. The ⟨error message⟩ must be short: it is cropped at the end of one line.

TeXhackers note: This function expands to an empty token list after two steps. Tokens inserted in response to TeX's prompt are read with the current category code setting, and inserted just after the place where the error message was issued.

8 Internal **l3msg** functions

The following functions are used in several kernel modules.

__msg_log_next:

New: 2015-08-05

__msg_log_next: ⟨show-command⟩

Causes the next ⟨show-command⟩ to send its output to the log file instead of the terminal. This allows for instance \cs_log:N to be defined as __msg_log_next: \cs_show:N. The effect of this command lasts until the next use of __msg_show_wrap:Nn or __msg_-show_wrap:n or __msg_show_variable:NNNnn, in other words until the next time the ε-TeX primitive \showtokens would have been used for showing to the terminal or until the next variable-not-defined error.

__msg_show_pre:nnnnnn
__msg_show_pre:(nnxxxx|nnnnnV)

New: 2015-08-05

__msg_show_pre:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩}
{⟨arg three⟩} {⟨arg four⟩}

Prints the ⟨message⟩ from ⟨module⟩ in the terminal (or log file if __msg_log_next: was issued) without formatting. Used in messages which print complex variable contents completely.

`\__msg_show_variable:NNNnn`	`\__msg_show_variable:NNNnn` ⟨variable⟩ ⟨if-exist⟩ ⟨if-empty⟩ {⟨msg⟩} {⟨formatted content⟩}
New: 2015-08-04	

If the ⟨variable⟩ does not exist according to ⟨if-exist⟩ (typically `\cs_if_exist:NTF`) then throw an error and do nothing more. Otherwise, if ⟨msg⟩ is not empty, display the message `LaTeX/kernel/show-`⟨msg⟩ with `\token_to_str:N` ⟨variable⟩ as a first argument, and a second argument that is `?` or empty depending on the result of ⟨if-empty⟩ (typically `\tl_if_empty:NTF`) on the ⟨variable⟩. Then display the ⟨formatted content⟩ by giving it as an argument to `\__msg_show_wrap:n`.

`\__msg_show_wrap:Nn`	`\__msg_show_wrap:Nn` ⟨function⟩ {⟨expression⟩}
New: 2015-08-03	
Updated: 2015-08-07	

Shows or logs the ⟨expression⟩ (turned into a string), an equal sign, and the result of applying the ⟨function⟩ to the {⟨expression⟩}. For instance, if the ⟨function⟩ is `\int_-eval:n` and the ⟨expression⟩ is `1+2` then this will log `> 1+2=3`. The case where the ⟨function⟩ is `\tl_to_str:n` is special: then the string representation of the ⟨expression⟩ is only logged once.

`\__msg_show_wrap:n`	`\__msg_show_wrap:n` {⟨formatted text⟩}
New: 2015-08-03	

Shows or logs the ⟨formatted text⟩. After expansion, unless it is empty, the ⟨formatted text⟩ must contain `>`, and the part of ⟨formatted text⟩ before the first `>` is removed. Failure to do so causes low-level TeX errors.

`\__msg_show_item:n`	`\__msg_show_item:n` ⟨item⟩
`\__msg_show_item:nn`	`\__msg_show_item:nn` ⟨item-key⟩ ⟨item-value⟩
`\__msg_show_item_unbraced:nn`	
Updated: 2012-09-09	

Auxiliary functions used within the last argument of `\__msg_show_variable:NNNnn` or `\__msg_show_wrap:n` to format variable items correctly for display. The `\__msg_show_-item:n` version is used for simple lists, the `\__msg_show_item:nn` and `\__msg_show_-item_unbraced:nn` versions for key–value like data structures.

`\c__msg_coding_error_text_tl`	

The text

    This is a coding error.

used by kernel functions when erroneous programming input is encountered.

Part XX
The **l3keys** package
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
  }
```

At a document level, `\keys_set:nn` will be 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`; spaces are *ignored* in key names. As will be discussed in section 2, it is suggested that the character `/` is reserved for sub-division of keys into logical groups. 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
  }
```

will create a key called `\l_mymodule_tmp_tl`, and not one called `key`.

1   Creating keys

`\keys_define:nn`

Updated: 2015-11-07

`\keys_define:nn {⟨module⟩} {⟨keyval list⟩}`

Parses the ⟨*keyval list*⟩ and defines the keys listed there for ⟨*module*⟩. The ⟨*module*⟩ name should be a text value, but there are no restrictions on the nature of the text. 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.*, will override one another. Some other properties are mutually exclusive, notably `.value_required:n` and `.value_forbidden:n`, and so will 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 with the exception of the special `.undefine:` property, all key properties will define the key within the current TeX scope.

.bool_set:N
.bool_set:c
.bool_gset:N
.bool_gset:c

Updated: 2013-07-08

⟨*key*⟩ `.bool_set:N` = ⟨*boolean*⟩

Defines ⟨*key*⟩ to set ⟨*boolean*⟩ 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*⟩

Defines ⟨*key*⟩ to set ⟨*boolean*⟩ to the logical inverse of ⟨*value*⟩ (which must be either `true` or `false`). If the ⟨*boolean*⟩ 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 3.

.choices:nn
.choices:(Vn|on|xn)

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 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 will be 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*⟩. The `x`-type variant will expand ⟨*code*⟩ at the point where the ⟨*key*⟩ is created.

.default:n

.default:(V|o|x)

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 will not trigger an error.

.dim_set:N
.dim_set:c
.dim_gset:N
.dim_gset:c

⟨*key*⟩ `.dim_set:N = `⟨*dimension*⟩

Defines ⟨*key*⟩ to set ⟨*dimension*⟩ to ⟨*value*⟩ (which must a dimension expression). If the variable does not exist, it will be created globally at the point that the key is set up.

.fp_set:N
.fp_set:c
.fp_gset:N
.fp_gset:c

⟨*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 will be created globally at the point that the key is set up.

.groups:n

New: 2013-07-14

⟨*key*⟩ `.groups:n = {`⟨*groups*⟩`}`

Defines ⟨*key*⟩ as belonging to the ⟨*groups*⟩ declared. Groups provide a "secondary axis" for selectively setting keys, and are described in Section 6.

.initial:n

.initial:(V|o|x)

Updated: 2013-07-09

⟨*key*⟩ `.initial:n = {`⟨*value*⟩`}`

Initialises the ⟨*key*⟩ with the ⟨*value*⟩, equivalent to

$$\text{\keys\_set:nn } \{⟨module⟩\} \ \{ \ ⟨key⟩ = ⟨value⟩ \ \}$$

.int_set:N
.int_set:c
.int_gset:N
.int_gset:c

⟨*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 will be created globally at the point that the key is set up.

.meta:n

Updated: 2013-07-10

⟨*key*⟩ `.meta:n = {`⟨*keyval list*⟩`}`

Makes ⟨*key*⟩ a meta-key, which will set ⟨*keyval list*⟩ in one go. If ⟨*key*⟩ is given with a value at the time the key is used, then the value will be passed through to the subsidiary ⟨*keys*⟩ for processing (as `#1`).

.meta:nn

New: 2013-07-10

⟨*key*⟩ .meta:nn = {⟨*path*⟩} {⟨*keyval list*⟩}

Makes ⟨*key*⟩ a meta-key, which will set ⟨*keyval list*⟩ in one go using the ⟨*path*⟩ in place of the current one. If ⟨*key*⟩ is given with a value at the time the key is used, then the value will be passed through to the subsidiary ⟨*keys*⟩ for processing (as #1).

.multichoice:

New: 2011-08-21

⟨*key*⟩ .multichoice:

Sets ⟨*key*⟩ to act as a multiple choice key. Each valid choice for ⟨*key*⟩ must then be created, as discussed in section 3.

.multichoices:nn
.multichoices:(Vn|on|xn)

New: 2011-08-21
Updated: 2013-07-10

⟨*key*⟩ .multichoices:nn {⟨*choices*⟩} {⟨*code*⟩}

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 3.

.skip_set:N
.skip_set:c
.skip_gset:N
.skip_gset:c

⟨*key*⟩ .skip_set:N = ⟨*skip*⟩

Defines ⟨*key*⟩ to set ⟨*skip*⟩ to ⟨*value*⟩ (which must be a skip expression). If the variable does not exist, it will be 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 = ⟨*token list variable*⟩

Defines ⟨*key*⟩ to set ⟨*token list variable*⟩ to ⟨*value*⟩. If the variable does not exist, it will be created globally at the point that the key is set up.

.tl_set_x:N
.tl_set_x:c
.tl_gset_x:N
.tl_gset_x:c

⟨*key*⟩ .tl_set_x:N = ⟨*token list variable*⟩

Defines ⟨*key*⟩ to set ⟨*token list variable*⟩ to ⟨*value*⟩, which will be subjected to an x-type expansion (*i.e.* using \tl_set:Nx). If the variable does not exist, it will be 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 scope.

.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 will cancel 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 will cancel the restriction.

2 Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several sub-groups for a given module. This can be achieved either by adding a sub-division to the module name:

```
\keys_define:nn { module / subgroup }
  { key .code:n = code }
```

or to the key name:

```
\keys_define:nn { mymodule }
  { subgroup / 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 `module/subgroup/key`.

As will be illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

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 dependant 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.

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 5. 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:nnxxx { 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 define as sub-keys. Thus both

```
\keys_define:nn { mymodule }
  {
    key .multichoices:nn =
      { choice-a, choice-b, choice-c }
```

166

```
    {
      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`.

4   Setting keys

\keys_set:nn
\keys_set:(nV|nv|no)

Updated: 2015-11-07

`\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 will be illustrated later.

`\l_keys_key_tl`
`\l_keys_path_tl`
`\l_keys_value_tl`

Updated: 2015-07-14

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 three token list variables. These may be used within the code of the key.

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`.

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_tl`, and will have been processed by `\tl_to_str:n`.

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_tl`, and will have been processed by `\tl_-to_str:n`.

5 Handling of unknown keys

If a key has not previously been defined (is unknown), `\keys_set:nn` will look 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.

```
\keys_define:nn { mymodule }
  {
    unknown .code:n =
      You~tried~to~set~key~'\l_keys_key_tl'~to~'#1'.
  }
```

`\keys_set_known:nnN`
`\keys_set_known:(nVN|nvN|noN)`
`\keys_set_known:nn`
`\keys_set_known:(nV|nv|no)`

New: 2011-08-23
Updated: 2015-11-07

`\keys_set_known:nnN` {⟨*module*⟩} {⟨*keyval list*⟩} ⟨*tl*⟩

In some cases, the desired behavior is to simply ignore unknown keys, collecting up information on these for later processing. The `\keys_set_known:nnN` 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. The key–value pairs for each *unknown* key name will be stored in the ⟨*tl*⟩ in a comma-separated form (*i.e.* an edited version of the ⟨*keyval list*⟩). The `\keys_set_known:nn` version skips this stage.

Use of `\keys_set_known:nnN` can be nested, with the correct residual ⟨*keyval list*⟩ returned at each stage.

6 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` will attempt 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          ,
  }
```

will assign `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_filter:nnnN`
`\keys_set_filter:(nnVN|nnvN|nnoN)`
`\keys_set_filter:nnn`
`\keys_set_filter:(nnV|nnv|nno)`

New: 2013-07-14
Updated: 2015-11-07

`\keys_set_filter:nnnN {⟨module⟩} {⟨groups⟩} {⟨keyval list⟩} ⟨tl⟩`

Actives key filtering in an "opt-out" sense: keys assigned to any of the ⟨*groups*⟩ specified will be ignored. The ⟨*groups*⟩ are given as a comma-separated list. Unknown keys are not assigned to any group and will thus always be set. The key–value pairs for each key which is filtered out will be stored in the ⟨*tl*⟩ in a comma-separated form (*i.e.* an edited version of the ⟨*keyval list*⟩). The `\keys_set_filter:nnn` version skips this stage.

Use of `\keys_set_filter:nnnN` can be nested, with the correct residual ⟨*keyval list*⟩ returned at each stage.

```
\keys_set_groups:nnn
\keys_set_groups:(nnV|nnv|nno)
```
New: 2013-07-14
Updated: 2015-11-07

`\keys_set_groups:nnn {⟨module⟩} {⟨groups⟩} {⟨keyval list⟩}`

Actives key filtering in an "opt-in" sense: only keys assigned to one or more of the ⟨*groups*⟩ specified will be set. The ⟨*groups*⟩ are given as a comma-separated list. Unknown keys are not assigned to any group and will thus never be set.

7 Utility functions for keys

```
\keys_if_exist_p:nn ⋆
\keys_if_exist:nnTF ⋆
```
Updated: 2015-11-07

`\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: 2015-11-07

`\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⟩}`

Shows the information associated to the ⟨*key*⟩ for a ⟨*module*⟩, including the function which is used to actually implement it.

8 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).

170

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 will have exactly
one set removed (after space trimming), thus

```
key = {value here},
```

and

```
key = value here,
```

are treated identically.

\keyval_parse:NNn

Updated: 2011-09-08

\keyval_parse:NNn ⟨function₁⟩ ⟨function₂⟩ {⟨key-value list⟩}

Parses the ⟨key–value list⟩ into a series of ⟨keys⟩ and associated ⟨values⟩, or keys alone
(if no ⟨value⟩ was given). ⟨function₁⟩ should take one argument, while ⟨function₂⟩
should absorb two arguments. After \keyval_parse:NNn has parsed the ⟨key–value list⟩,
⟨function₁⟩ will be used to process keys given with no value and ⟨function₂⟩ will be used
to process keys given with a value. The order of the ⟨keys⟩ in the ⟨key–value list⟩ will be
preserved. Thus

```
\keyval_parse:NNn \function:n \function:nn
  { key1 = value1 , key2 = value2, key3 = , key4 }
```

will be 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 noth-
ing) 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.

Part XXI

The **l3file** package
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 will attempt to locate them 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. Any active characters (as declared in `\l_char_active_seq`) will *not* be expanded, allowing the direct use of these in file names. File names will be quoted using `"` tokens if they contain spaces: as a result, `"` tokens are *not* permitted in file names.

1   File operation functions

`\g_file_current_name_tl`

Contains the name of the current LaTeX file. This variable should not be modified: it is intended for information only. It will be equal to `\c_sys_jobname_str` at the start of a LaTeX run and will be modified each time a file is read using `\file_input:n`.

`\file_if_exist:n`*TF*

Updated: 2012-02-10

`\file_if_exist:nTF {`⟨*file name*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Searches for ⟨*file name*⟩ using the current TeX search path and the additional paths controlled by `\file_path_include:n`).

`\file_add_path:nN`

Updated: 2012-02-10

`\file_add_path:nN {`⟨*file name*⟩`}` ⟨*tl var*⟩

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. If the file is not found then the ⟨*tl var*⟩ will contain the marker `\q_no_value`.

`\file_input:n`

Updated: 2012-02-17

`\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 will be raised if the file is not found.

`\file_path_include:n`

Updated: 2012-07-04

`\file_path_include:n {`⟨*path*⟩`}`

Adds ⟨*path*⟩ to the list of those used to search when reading files. The assignment is local. The ⟨*path*⟩ is processed in the same way as a ⟨*file name*⟩, *i.e.*, with `x`-type expansion except active characters.

`\file_path_remove:n`

Updated: 2012-07-04

`\file_path_remove:n {⟨path⟩}`

Removes ⟨path⟩ from the list of those used to search when reading files. The assignment is local. The ⟨path⟩ is processed in the same way as a ⟨file name⟩, *i.e.*, with x-type expansion except active characters.

`\file_list:`

`\file_list:`

This function will list all files loaded using `\file_input:n` in the log file.

1.1 Input–output stream management

As TEX is limited to 16 input streams and 16 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 a `\ior_-close:N` instruction is given or the TEX run ends.

`\ior_open:NnTF`
`\ior_open:cnTF`

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:cn`

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 will clear any existing content in the file (*i.e.* writing is *not* additive).

`\ior_close:N`	`\ior_close:N` ⟨stream⟩
`\ior_close:c`	`\iow_close:N` ⟨stream⟩
`\iow_close:N`	
`\iow_close:c`	
Updated: 2012-07-31	

Closes the ⟨stream⟩. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.

`\ior_list_streams:`	`\ior_list_streams:`
`\iow_list_streams:`	`\iow_list_streams:`
Updated: 2015-08-01	

Displays a list of the file names associated with each open stream: intended for tracking down problems.

1.2 Reading from files

`\ior_get:NN`	`\ior_get:NN` ⟨stream⟩ ⟨token list variable⟩
New: 2012-06-24	

Function that reads one or more lines (until an equal number of left and right braces are found) from the input ⟨stream⟩ and stores the result locally in the ⟨token list⟩ variable. If the ⟨stream⟩ is not open, input is requested from the terminal. The material read from the ⟨stream⟩ will be tokenized by TEX according to the category codes in force when the function is used. Note that any blank lines will be converted to the token `\par`. Therefore, if skipping blank lines is requires a test such as

```
\ior_get:NN \l_my_stream \l_tmpa_tl
\tl_set:Nn \l_tmpb_tl { \par }
\tl_if_eq:NNF \l_tmpa_tl \l_tmpb_tl
...
```

may be used. Also notice that if multiple lines are read to match braces then the resulting token list will contain `\par` tokens. As normal TEX tokenization is in force, any lines which do not end in a comment character (usually `%`) will have the line ending converted to a space, so for example input

```
a b  c
```

will result in a token list `a` `b` `c` .

TEXhackers note: This protected macro expands to the TEX primitive `\read` along with the `to` keyword.

\ior_get_str:NN

New: 2012-06-24
Updated: 2012-07-31

`\ior_get_str:NN` ⟨stream⟩ ⟨token list variable⟩

Function that reads one line from the input ⟨*stream*⟩ and stores the result locally in the ⟨*token list*⟩ variable. If the ⟨*stream*⟩ is not open, input is requested from the terminal. 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 will always only read one line and any blank lines in the input will result in the ⟨*token list variable*⟩ being empty. Unlike `\ior_get:NN`, line ends do not receive any special treatment. Thus input

    a b c

will result in a token list `a b  c` with the letters `a`, `b`, and `c` having category code 12.

TEXhackers note: This protected macro is a wrapper around the ε-TEX primitive `\readline`. However, the end-line character normally added by this primitive is not included in the result of `\ior_get_str:NN`.

\ior_if_eof_p:N ⋆
\ior_if_eof:NTF ⋆

Updated: 2012-02-10

`\ior_if_eof_p:N` ⟨stream⟩
`\ior_if_eof:NTF` ⟨stream⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the end of a ⟨*stream*⟩ has been reached during a reading operation. The test will also return a `true` value if the ⟨*stream*⟩ is not open.

2 Writing to files

\iow_now:Nn
\iow_now:(Nx|cn|cx)

Updated: 2012-06-05

`\iow_now:Nn` ⟨stream⟩ {⟨tokens⟩}

This functions 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:x

`\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:x

`\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:(Nx|cn|cx)

`\iow_shipout:Nn` ⟨stream⟩ {⟨tokens⟩}

This functions writes ⟨*tokens*⟩ to the specified ⟨*stream*⟩ when the current page is finalised (*i.e.* at shipout). The x-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_-x: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` will not be recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additionnal unwanted line-breaks.

\iow_shipout_x:Nn
\iow_shipout_x:(Nx|cn|cx)

Updated: 2012-09-08

\iow_shipout_x:Nn ⟨*stream*⟩ {⟨*tokens*⟩}

This functions 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 will not be recognized in the argument of \iow_shipout:Nn. This may lead to the insertion of additionnal 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:Nx \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: will not be recognized by TEX, which may lead to the insertion of additionnal unwanted line-breaks. This issue only affects \iow_shipout:Nn, \iow_shipout_x:Nn and direct uses of primitive operations.

2.1 Wrapping lines in output

\iow_wrap:nnnN

New: 2012-06-28
Updated: 2015-08-05

\iow_wrap:nnnN {⟨text⟩} {⟨run-on text⟩} {⟨set up⟩} ⟨function⟩

This function will wrap the ⟨text⟩ to a fixed number of characters per line. At the start of each line which is wrapped, the ⟨run-on text⟩ will be inserted. The line character count targeted will be 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:

- \\ 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_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 will typically be a wrapper around a write operation. The output of \iow_wrap:nnnN (*i.e.* the argument passed to the ⟨function⟩) will consist of characters of category "other" (category code 12), with the exception of spaces which will have category "space" (category code 10). This means that the output will *not* expand further when written to a file.

TEXhackers note: Internally, \iow_wrap:nnnN carries out an x-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_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 will 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 TEXlive and MiKTEX systems.

\c_catcode_other_space_tl

New: 2011-09-05

Token list containing one character with category code 12, ("other"), and character code 32 (space).

177

2.2 Constant input–output streams

\c_term_ior

Constant input stream for reading from the terminal. Reading from this stream using `\ior_get:NN` or similar will result in a prompt from TeX of the form

    <tl>=

\c_log_iow
\c_term_iow

Constant output streams for writing to the log and to the terminal (plus the log), respectively.

2.3 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`.

2.4 Internal file functions and variables

\g__file_internal_ior

Used to test for the existence of files when opening.

\l__file_internal_name_tl

Used to return the full name of a file for internal use. This is set by `\file_if_exist:nTF` and `\__file_if_exist:nT`, and the value may then be used to load a file directly provided no further operations intervene.

__file_name_sanitize:nn

New: 2012-02-09

`\__file_name_sanitize:nn {⟨name⟩} {⟨tokens⟩}`

Exhaustively-expands the ⟨name⟩ with the exception of any category ⟨active⟩ (catcode 13) tokens, which are not expanded. The list of ⟨active⟩ tokens is taken from `\l_char_-active_seq`. The ⟨sanitized name⟩ is then inserted (in braces) after the ⟨tokens⟩, which should further process the file name. If any spaces are found in the name after expansion, an error is raised.

2.5 Internal input–output functions

__ior_open:Nn
__ior_open:No

New: 2012-01-23

`\__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 `\file_add_path:nN`,

178

`\__iow_with:Nnn`

New: 2014-08-23

`\__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.

Part XXII
The **l3fp** package: 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 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: $x < y$, $x <= y$, $x >? y$, $x \mathbin{!} = y$ *etc.*

- Boolean logic: negation $!\,x$, conjunction $x \mathbin{\&\&} y$, disjunction $x \mathbin{||} y$, ternary operator $x\,?\,y\,{:}\,z$.

- Random numbers: $rand()$, $randint(m,n)$.

- Exponentials: $\exp x$, $\ln x$, x^y.

- Trigonometry: $\sin x$, $\cos x$, $\tan x$, $\cot x$, $\sec x$, $\csc x$ expecting their arguments in radians, and sind x, cosd x, tand x, cotd x, secd x, cscd x expecting their arguments in degrees.

- Inverse trigonometric functions: asin x, acos x, atan x, acot x, asec x, acsc x giving a result in radians, and asind x, acosd x, atand x, acotd x, asecd x, acscd x giving a result in degrees.

(not yet) Hyperbolic functions and their inverse functions: $\sinh x$, $\cosh x$, $\tanh x$, $\coth x$, sech x, csch, and asinh x, acosh x, atanh x, acoth x, asech x, acsch x.

- Extrema: $\max(x, y, \ldots)$, $\min(x, y, \ldots)$, $\mathrm{abs}(x)$.

- Rounding functions ($n = 0$ by default, $t = \texttt{NaN}$ by default): $\mathrm{trunc}(x, n)$ rounds towards zero, $\mathrm{floor}(x, n)$ rounds towards $-\infty$, $\mathrm{ceil}(x, n)$ rounds towards $+\infty$, $\mathrm{round}(x, n, t)$ 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".

- 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 points, expressing dimensions in points and ignoring the stretch and shrink components of skips.

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. See section 9.1 for a description of what a floating point is, section 9.2 for details about how an expression is parsed, and section 9.3 to know what the various operations do. Some operations may raise exceptions (error messages), described in section 7.

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} $.
```

But in all fairness, this 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.

```
\usepackage{xparse, siunitx}
\ExplSyntaxOn
\NewDocumentCommand { \calcnum } { m }
  { \num { \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\calcnum { 2 pi * sin ( 2.3 ^ 5 ) }
```

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⟩ will initially be +0.

\fp_const:Nn
\fp_const:cn

Updated: 2012-05-08

\fp_const:Nn ⟨fp var⟩ {⟨floating point expression⟩}

Creates a new constant ⟨fp var⟩ or raises an error if the name is already taken. The ⟨fp var⟩ will be set globally equal to the result of evaluating the ⟨floating point expression⟩.

\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.

2 Setting floating point variables

\fp_set:Nn
\fp_set:cn
\fp_gset:Nn
\fp_gset:cn

Updated: 2012-05-08

\fp_set:Nn ⟨fp var⟩ {⟨floating point expression⟩}

Sets ⟨fp var⟩ equal to the result of computing the ⟨floating point expression⟩.

\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 var₁*⟩ ⟨*fp var₂*⟩

Sets the floating point variable ⟨*fp var₁*⟩ equal to the current value of ⟨*fp var₂*⟩.

\fp_add:Nn
`\fp_add:cn`
\fp_gadd:Nn
`\fp_gadd:cn`

Updated: 2012-05-08

`\fp_add:Nn` ⟨*fp var*⟩ {⟨*floating point expression*⟩}

Adds the result of computing the ⟨*floating point expression*⟩ to the ⟨*fp var*⟩.

\fp_sub:Nn
`\fp_sub:cn`
\fp_gsub:Nn
`\fp_gsub:cn`

Updated: 2012-05-08

`\fp_sub:Nn` ⟨*fp var*⟩ {⟨*floating point expression*⟩}

Subtracts the result of computing the ⟨*floating point expression*⟩ from the ⟨*fp var*⟩.

3 Using floating point numbers

\fp_eval:n ⋆

New: 2012-05-08
Updated: 2012-07-08

`\fp_eval:n` {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩ 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. This function is identical to `\fp_to_decimal:n`.

\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` {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩ 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.

\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` {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩ 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. 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 {`⟨*floating point expression*⟩`}`

Evaluates the ⟨*floating point expression*⟩, 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. 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 {`⟨*floating point expression*⟩`}`

Evaluates the ⟨*floating point expression*⟩ and expresses the result in scientific notation:

$$\langle \textit{optional } \texttt{-}\rangle\langle \textit{digit}\rangle\texttt{.}\langle \textit{15 digits}\rangle\texttt{e}\langle \textit{optional sign}\rangle\langle \textit{exponent}\rangle$$

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).

\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 {`⟨*floating point expression*⟩`}`

Evaluates the ⟨*floating point expression*⟩ 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 (see `\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, will be made up of letters.

\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. This function is identical to `\fp_to_decimal:N`.

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 ⋆

Updated: 2012-05-08

`\fp_compare_p:nNn {⟨fpexpr₁⟩} ⟨relation⟩ {⟨fpexpr₂⟩}`
`\fp_compare:nNnTF {⟨fpexpr₁⟩} ⟨relation⟩ {⟨fpexpr₂⟩} {⟨true code⟩} {⟨false code⟩}`

Compares the ⟨*fpexpr₁*⟩ and the ⟨*fpexpr₂*⟩, and returns `true` if the ⟨*relation*⟩ is obeyed. Two floating point numbers x and y may obey four mutually exclusive relations: $x\langle y, x=y, x\rangle y$, or x and y are not ordered. The latter case occurs exactly when either operand is `NaN`, and this relation is denoted by the symbol `?`. 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
```

\fp_compare_p:n ⋆
\fp_compare:nTF ⋆

Updated: 2012-12-14

```
\fp_compare_p:n
  {
    ⟨fpexpr₁⟩ ⟨relation₁⟩
    ...
    ⟨fpexpr_N⟩ ⟨relation_N⟩
    ⟨fpexpr_{N+1}⟩
  }
\fp_compare:nTF
  {
    ⟨fpexpr₁⟩ ⟨relation₁⟩
    ...
    ⟨fpexpr_N⟩ ⟨relation_N⟩
    ⟨fpexpr_{N+1}⟩
  }
  {⟨true code⟩} {⟨false code⟩}
```

Evaluates the ⟨*floating point expressions*⟩ as described for `\fp_eval:n` and compares consecutive result using the corresponding ⟨*relation*⟩, namely it compares ⟨*intexpr₁*⟩ and ⟨*intexpr₂*⟩ using the ⟨*relation₁*⟩, then ⟨*intexpr₂*⟩ and ⟨*intexpr₃*⟩ using the ⟨*relation₂*⟩, until finally comparing ⟨*intexpr_N*⟩ and ⟨*intexpr_{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 ⟨*floating point expressions*⟩ are computed, even if one comparison is `false`. Two floating point numbers x and y may obey four mutually exclusive relations: $x\langle y, x=y, x\rangle y$, or x and y are not ordered. The latter case occurs exactly when one of the operands is `NaN`, and this relation is denoted by the symbol `?`. 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 floatin 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).

5 Floating point expression loops

\fp_do_until:nNnn ☆

New: 2012-08-16

\fp_do_until:nNnn {⟨fpexpr₁⟩} ⟨relation⟩ {⟨fpexpr₂⟩} {⟨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⟩ will be inserted into the input stream again and a loop will occur until the ⟨relation⟩ is true.

\fp_do_while:nNnn ☆

New: 2012-08-16

\fp_do_while:nNnn {⟨fpexpr₁⟩} ⟨relation⟩ {⟨fpexpr₂⟩} {⟨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 true then the ⟨code⟩ will be inserted into the input stream again and a loop will occur until the ⟨relation⟩ is false.

\fp_until_do:nNnn ☆

New: 2012-08-16

\fp_until_do:nNnn {⟨fpexpr₁⟩} ⟨relation⟩ {⟨fpexpr₂⟩} {⟨code⟩}

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 will be repeated, and a loop will occur until the test is true.

\fp_while_do:nNnn ☆

New: 2012-08-16

\fp_while_do:nNnn {⟨fpexpr₁⟩} ⟨relation⟩ {⟨fpexpr₂⟩} {⟨code⟩}

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 will be repeated, and a loop will occur until the test is false.

\fp_do_until:nn ☆

New: 2012-08-16

\fp_do_until:nn { ⟨fpexpr₁⟩ ⟨relation⟩ ⟨fpexpr₂⟩ } {⟨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:nTF. If the test is false then the ⟨code⟩ will be inserted into the input stream again and a loop will occur until the ⟨relation⟩ is true.

\fp_do_while:nn ☆

New: 2012-08-16

\fp_do_while:nn { ⟨fpexpr₁⟩ ⟨relation⟩ ⟨fpexpr₂⟩ } {⟨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:nTF. If the test is true then the ⟨code⟩ will be inserted into the input stream again and a loop will occur until the ⟨relation⟩ is false.

\fp_until_do:nn ☆

New: 2012-08-16

\fp_until_do:nn { ⟨fpexpr₁⟩ ⟨relation⟩ ⟨fpexpr₂⟩ } {⟨code⟩}

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 will be repeated, and a loop will occur until the test is true.

\fp_while_do:nn ☆

New: 2012-08-16

\fp_while_do:nn { ⟨*fpexpr₁*⟩ ⟨*relation*⟩ ⟨*fpexpr₂*⟩ } {⟨*code*⟩}

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 will be repeated, and a loop will occur until the test is false.

\fp_step_function:nnnN ☆

New: 2016-11-21

\fp_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 floating point 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 }
\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]

\fp_step_inline:nnnn

New: 2016-11-21

\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. 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).

6   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_e_fp

Updated: 2012-05-08

The value of the base of the natural logarithm, e = exp(1).

186

`\c_pi_fp`	The value of π. This can be input directly in a floating point expression as `pi`.
Updated: 2013-11-17	

`\c_one_degree_fp`	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`.
New: 2012-05-08	
Updated: 2013-11-17	

`\l_tmpa_fp`	Scratch floating points for local assignment. These are never used by the kernel code, and
`\l_tmpb_fp`	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`	Scratch floating points for global assignment. These are never used by the kernel code,
`\g_tmpb_fp`	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.

7 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 IEEE standard defines 5 types of exceptions.

- *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 almost any operation involving a `NaN`. This normally results in a `NaN`, except for conversion functions whose target type does not have a notion of `NaN` (*e.g.*, `\fp_to_dim:n`).

- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating *e.g.*, $\ln(0)$ or $\cot(0)$. This results in $\pm\infty$.

- *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 is associated a "flag", which can be either *on* or *off*. By default, the "invalid operation" exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions only raise the corresponding flag. The state of the flag can be tested and modified. The behaviour when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and turn the flag on, or only turn the flag on, or do nothing at all.

`\fp_if_flag_on_p:n` *★*	`\fp_if_flag_on_p:n {⟨exception⟩}`
`\fp_if_flag_on:nTF` *★*	`\fp_if_flag_on:nTF {⟨exception⟩} {⟨true code⟩} {⟨false code⟩}`
New: 2012-08-08	

Tests if the flag for the ⟨*exception*⟩ is on, which normally means the given ⟨*exception*⟩ has occurred. *This function is experimental, and may be altered or removed.*

`\fp_flag_off:n`	`\fp_flag_off:n {⟨exception⟩}`
New: 2012-08-08	

Locally turns off the flag which indicates whether the ⟨*exception*⟩ has occurred. *This function is experimental, and may be altered or removed.*

`\fp_flag_on:n` *★*	`\fp_flag_on:n {⟨exception⟩}`
New: 2012-08-08	

Locally turns on the flag to indicate (or pretend) that the ⟨*exception*⟩ has occurred. Note that this function is expandable: it is used internally by l3fp to signal when exceptions do occur. *This function is experimental, and may be altered or removed.*

`\fp_trap:nn`	`\fp_trap:nn {⟨exception⟩} {⟨trap type⟩}`
New: 2012-07-19	
Updated: 2012-08-08	

All occurrences of the ⟨*exception*⟩ (`invalid_operation`, `division_by_zero`, `overflow`, or `underflow`) 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.

This function is experimental, and may be altered or removed.

8 Viewing floating points

`\fp_show:N`	`\fp_show:N ⟨fp var⟩`
`\fp_show:c`	`\fp_show:n {⟨floating point expression⟩}`
`\fp_show:n`	
New: 2012-05-08	
Updated: 2015-08-07	

Evaluates the ⟨*floating point expression*⟩ and displays the result in the terminal.

9 Floating point expressions

9.1 Input of floating point numbers

We support four types of floating point numbers:

- $\pm 0.d_1 d_2 \ldots d_{16} \cdot 10^n$, a normal floating point number, with $d_i \in [0,9]$, $d_1 \neq 0$, and $|n| \leq 10000$;

- ± 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);

(not yet) subnormal numbers $\pm 0.d_1 d_2 \ldots d_{16} \cdot 10^{-10000}$ with $d_1 = 0$.

Normal floating point numbers are stored in base 10, with 16 significant figures.

On input, a normal floating point number consists of:

- $\langle sign \rangle$: a possibly empty string of `+` and `-` characters;

- $\langle significand \rangle$: a non-empty string of digits together with zero or one dot;

- $\langle exponent \rangle$ optionally: the character `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 $\langle sign \rangle$ contains an even number of `-`, and `-` otherwise, hence, an empty $\langle sign \rangle$ denotes a non-negative input. The stored significand is obtained from $\langle significand \rangle$ 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 $\langle significand \rangle$ has at most 16 digits. The stored $\langle exponent \rangle$ is obtained by combining the input $\langle exponent \rangle$ (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 $\langle exponent \rangle$ 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 `0` characters, and an optional `.` character), 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.

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 will be ignored.

- Any unrecognizable string triggers an error, and produces a `NaN`.

Note that `e-1` is not a representation of 10^{-1}, because it could be mistaken with the difference of "`e`" and 1. This is consistent with several other programming languages. However, in order to avoid confusions, `e-1` is not considered to be this difference either. To input the base of natural logarithms, use `exp(1)` or `\c_e_fp`.

9.2 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 `+`, `-`, `!`.

- Binary `*`, `/`, and implicit multiplication by juxtaposition (`2pi`, `3(4+5)`, *etc*).

- Binary `+` and `-`.

- Comparisons `>=`, `!=`, `<?`, *etc.*

- Logical `and`, denoted by `&&`.

- Logical `or`, denoted by `||`.

- Ternary operator `?:` (right associative).

The precedence of operations can be overridden using parentheses. In particular, those precedences imply that

$$\texttt{sin2pi} = \sin(2\pi) = 0,$$

$$\texttt{2\^{}2max(3,4)} = 2^{2\max(3,4)} = 256.$$

Functions are called on the value of their argument, contrarily to TeX macros.

9.3 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`.

?: `\fp_eval:n { ⟨operand₁⟩ ? ⟨operand₂⟩ : ⟨operand₃⟩ }`

The ternary operator `?:` results in ⟨*operand₂*⟩ if ⟨*operand₁*⟩ is true, and ⟨*operand₃*⟩ if it is false (equal to ± 0). All three ⟨*operands*⟩ are evaluated in all cases. 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 ⟨*operand₁*⟩ is true (non-zero), use that value, otherwise the value of ⟨*operand₂*⟩. Both ⟨*operands*⟩ are evaluated in all cases.

&& `\fp_eval:n { ⟨operand₁⟩ && ⟨operand₂⟩ }`

If ⟨*operand₁*⟩ is false (equal to ± 0), use that value, otherwise the value of ⟨*operand₂*⟩. Both ⟨*operands*⟩ are evaluated in all cases.

`<`	`\fp_eval:n`
`=`	`  {`
`>`	`    `$\langle operand_1\rangle$ $\langle relation_1\rangle$
`?`	`    ...`
	`    `$\langle operand_N\rangle$ $\langle relation_N\rangle$
Updated: 2013-12-14	`    `$\langle operand_{N+1}\rangle$
	`  }`

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 $\langle operand_i\rangle$ $\langle relation_j\rangle$ $\langle operand_{i+1}\rangle$ are true, and $+0$ otherwise. All ⟨*operands*⟩ are evaluated in all cases. See `\fp_compare:nTF` for details.

`+`	`\fp_eval:n { `$\langle operand_1\rangle$` + `$\langle operand_2\rangle$` }`
`-`	`\fp_eval:n { `$\langle operand_1\rangle$` - `$\langle operand_2\rangle$` }`

Computes the sum or the difference of its two ⟨*operands*⟩. The "invalid operation" exception occurs for $\infty - \infty$. "Underflow" and "overflow" occur when appropriate.

`*`	`\fp_eval:n { `$\langle operand_1\rangle$` * `$\langle operand_2\rangle$` }`
`/`	`\fp_eval:n { `$\langle operand_1\rangle$` / `$\langle operand_2\rangle$` }`

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.

`+`	`\fp_eval:n { + `$\langle operand\rangle$` }`
`-`	`\fp_eval:n { - `$\langle operand\rangle$` }`
`!`	`\fp_eval:n { ! `$\langle operand\rangle$` }`

The unary `+` does nothing, the unary `-` changes the sign of the ⟨*operand*⟩, and `!` ⟨*operand*⟩ evaluates to 1 if ⟨*operand*⟩ is false and 0 otherwise (this is the `not` boolean function). Those operations never raise exceptions.

`**`	`\fp_eval:n { `$\langle operand_1\rangle$` ** `$\langle operand_2\rangle$` }`
`^`	`\fp_eval:n { `$\langle operand_1\rangle$` ^ `$\langle operand_2\rangle$` }`

Raises ⟨*operand₁*⟩ to the power ⟨*operand₂*⟩. This operation is right associative, hence `2 ** 2 ** 3` equals $2^{2^3} = 256$. The "invalid operation" exception occurs if ⟨*operand₁*⟩ is negative or -0, and ⟨*operand₂*⟩ is not an integer, unless the result is zero (in that case, the sign is chosen arbitrarily to be $+0$). "Division by zero" occurs when raising ± 0 to a strictly negative power. "Underflow" and "overflow" occur when appropriate.

`abs`	`\fp_eval:n { abs( `$\langle fpexpr\rangle$` ) }`

Computes the absolute value of the ⟨*fpexpr*⟩. This function does not raise any exception beyond those raised when computing its operand ⟨*fpexpr*⟩. See also `\fp_abs:n`.

`exp`	`\fp_eval:n { exp( `$\langle fpexpr\rangle$` ) }`

Computes the exponential of the ⟨*fpexpr*⟩. "Underflow" and "overflow" occur when appropriate.

ln `\fp_eval:n { ln( ⟨fpexpr⟩ ) }`

Computes the natural logarithm of the ⟨*fpexpr*⟩. 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.

max `\fp_eval:n { max( ⟨fpexpr₁⟩ , ⟨fpexpr₂⟩ , ... ) }`
min `\fp_eval:n { min( ⟨fpexpr₁⟩ , ⟨fpexpr₂⟩ , ... ) }`

Evaluates each ⟨*fpexpr*⟩ and computes the largest (smallest) of those. If any of the ⟨*fpexpr*⟩ is a `NaN`, the result is `NaN`. Those operations do not raise exceptions.

round `\fp_eval:n { round ( ⟨fpexpr⟩ ) }`
trunc `\fp_eval:n { round ( ⟨fpexpr₁⟩ , ⟨fpexpr₂⟩ ) }`
ceil `\fp_eval:n { round ( ⟨fpexpr₁⟩ , ⟨fpexpr₂⟩ , ⟨fpexpr₃⟩ ) }`
floor

New: 2013-12-14
Updated: 2015-08-08

Only `round` accepts a third argument. Evaluates $⟨fpexpr_1⟩ = x$ and $⟨fpexpr_2⟩ = n$ and $⟨fpexpr_3⟩ = t$ 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 is neither $\pm\infty$ nor an integer, then an "invalid operation" exception is raised. When ⟨*fpexpr₂*⟩ is omitted, $n = 0$, *i.e.*, ⟨*fpexpr₁*⟩ 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`, or the deprecated `round-`, yields the largest multiple of 10^{-n} smaller or equal to x ("round towards negative infinity");

- `ceil`, or the deprecated `round+`, yields the smallest multiple of 10^{-n} greater or equal to x ("round towards positive infinity");

- `trunc`, or the deprecated `round0`, yields a multiple of 10^{-n} with the same sign as x and with the largest absolute value less that that of x ("round towards zero").

"Overflow" occurs if x is finite and the result is infinite (this can only happen if $⟨fpexpr_2⟩ < -9984$).

sin	`\fp_eval:n { sin( ⟨fpexpr⟩ ) }`
cos	`\fp_eval:n { cos( ⟨fpexpr⟩ ) }`
tan	`\fp_eval:n { tan( ⟨fpexpr⟩ ) }`
cot	`\fp_eval:n { cot( ⟨fpexpr⟩ ) }`
csc	`\fp_eval:n { csc( ⟨fpexpr⟩ ) }`
sec	`\fp_eval:n { sec( ⟨fpexpr⟩ ) }`

Updated: 2013-11-17 — Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the ⟨fpexpr⟩ 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 analog $\sin d(8 \times 180)$ 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.

sind	`\fp_eval:n { sind( ⟨fpexpr⟩ ) }`
cosd	`\fp_eval:n { cosd( ⟨fpexpr⟩ ) }`
tand	`\fp_eval:n { tand( ⟨fpexpr⟩ ) }`
cotd	`\fp_eval:n { cotd( ⟨fpexpr⟩ ) }`
cscd	`\fp_eval:n { cscd( ⟨fpexpr⟩ ) }`
secd	`\fp_eval:n { secd( ⟨fpexpr⟩ ) }`

New: 2013-11-02 — Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the ⟨fpexpr⟩ 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 analog $\sin d(8 \times 180)$ 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.

asin	`\fp_eval:n { asin( ⟨fpexpr⟩ ) }`
acos	`\fp_eval:n { acos( ⟨fpexpr⟩ ) }`
acsc	`\fp_eval:n { acsc( ⟨fpexpr⟩ ) }`
asec	`\fp_eval:n { asec( ⟨fpexpr⟩ ) }`

New: 2013-11-02 — Computes the arcsine, arccosine, arccosecant, or arcsecant of the ⟨fpexpr⟩ 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.

asind	`\fp_eval:n { asind( ⟨fpexpr⟩ ) }`
acosd	`\fp_eval:n { acosd( ⟨fpexpr⟩ ) }`
acscd	`\fp_eval:n { acscd( ⟨fpexpr⟩ ) }`
asecd	`\fp_eval:n { asecd( ⟨fpexpr⟩ ) }`

New: 2013-11-02 — Computes the arcsine, arccosine, arccosecant, or arcsecant of the ⟨fpexpr⟩ and returns the result in degrees, in the range $[-90, 90]$ for `asin` and `acsc` and $[0, 180]$ for `acos` and `asec`. For a result in radians, use `asin`, *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.

atan
acot
 	
New: 2013-11-02
	`\fp_eval:n { atan( ⟨fpexpr⟩ ) }`
	`\fp_eval:n { atan( ⟨fpexpr₁⟩ , ⟨fpexpr₂⟩ ) }`
	`\fp_eval:n { acot( ⟨fpexpr⟩ ) }`
	`\fp_eval:n { acot( ⟨fpexpr₁⟩ , ⟨fpexpr₂⟩ ) }`

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 ⟨*fpexpr*⟩: 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 $(\langle \mathit{fpexpr_2} \rangle, \langle \mathit{fpexpr_1} \rangle)$: this is the arctangent of $\langle \mathit{fpexpr_1} \rangle / \langle \mathit{fpexpr_2} \rangle$, possibly shifted by π depending on the signs of $\langle \mathit{fpexpr_1} \rangle$ and $\langle \mathit{fpexpr_2} \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle \mathit{fpexpr_1} \rangle, \langle \mathit{fpexpr_2} \rangle)$, equal to the arccotangent of $\langle \mathit{fpexpr_1} \rangle / \langle \mathit{fpexpr_2} \rangle$, possibly shifted by π. Both two-argument functions take values in the wider range $[-\pi, \pi]$. The ratio $\langle \mathit{fpexpr_1} \rangle / \langle \mathit{fpexpr_2} \rangle$ 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. Only the "underflow" exception can occur.

atand
acotd
 	
New: 2013-11-02
	`\fp_eval:n { atand( ⟨fpexpr⟩ ) }`
	`\fp_eval:n { atand( ⟨fpexpr₁⟩ , ⟨fpexpr₂⟩ ) }`
	`\fp_eval:n { acotd( ⟨fpexpr⟩ ) }`
	`\fp_eval:n { acotd( ⟨fpexpr₁⟩ , ⟨fpexpr₂⟩ ) }`

Those functions yield an angle in degrees: `atand` and `acotd` are their analogs in radians. The one-argument versions compute the arctangent or arccotangent of the ⟨*fpexpr*⟩: 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 $(\langle \mathit{fpexpr_2} \rangle, \langle \mathit{fpexpr_1} \rangle)$: this is the arctangent of $\langle \mathit{fpexpr_1} \rangle / \langle \mathit{fpexpr_2} \rangle$, possibly shifted by 180 depending on the signs of $\langle \mathit{fpexpr_1} \rangle$ and $\langle \mathit{fpexpr_2} \rangle$. The two-argument arccotangent computes the angle in polar coordinates of the point $(\langle \mathit{fpexpr_1} \rangle, \langle \mathit{fpexpr_2} \rangle)$, equal to the arccotangent of $\langle \mathit{fpexpr_1} \rangle / \langle \mathit{fpexpr_2} \rangle$, possibly shifted by 180. Both two-argument functions take values in the wider range $[-180, 180]$. The ratio $\langle \mathit{fpexpr_1} \rangle / \langle \mathit{fpexpr_2} \rangle$ 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. Only the "underflow" exception can occur.

sqrt New: 2013-12-14	`\fp_eval:n { sqrt( ⟨fpexpr⟩ ) }`

Computes the square root of the ⟨*fpexpr*⟩. The "invalid operation" is raised when the ⟨*fpexpr*⟩ is negative; 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: 2016-11-20	`\fp_eval:n { rand() }`

Produces a pseudo-random floating-point number (multiple of 10^{-16}) between 0 included and 1 excluded.

TₑXhackers note: This is a wrapper around `\pdfuniformdeviate`. The random seed can be queried using `\pdfrandomseed` and set using `\pdfsetrandomseed`.

randint	`\fp_eval:n { randint( ⟨fpexpr⟩ ) }`
New: 2016-11-20	`\fp_eval:n { randint( ⟨fpexpr₁⟩ , ⟨fpexpr₂⟩ ) }`

Produces a pseudo-random integer between 1 and ⟨*fpexpr*⟩ or between ⟨*fpexpr₁*⟩ and ⟨*fpexpr₂*⟩ inclusive. The bounds must be integers in the range $(-10^{16}, 10^{16})$ and the first must be smaller or equal to the second.

TEXhackers note: This is a wrapper around `\pdfuniformdeviate`. The random seed can be queried using `\pdfrandomseed` and set using `\pdfsetrandomseed`.

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

$$1\text{in} = 72.27\text{pt}$$
$$1\text{pt} = 1\text{pt}$$
$$1\text{pc} = 12\text{pt}$$
$$1\text{cm} = \frac{1}{2.54}\text{in} = 28.45275590551181\text{pt}$$
$$1\text{mm} = \frac{1}{25.4}\text{in} = 2.845275590551181\text{pt}$$
$$1\text{dd} = 0.376065\text{mm} = 1.07000856496063\text{pt}$$
$$1\text{cc} = 12\text{dd} = 12.84010277952756\text{pt}$$
$$1\text{nd} = 0.375\text{mm} = 1.066978346456693\text{pt}$$
$$1\text{nc} = 12\text{nd} = 12.80374015748031\text{pt}$$
$$1\text{bp} = \frac{1}{72}\text{in} = 1.00375\text{pt}$$
$$1\text{sp} = 2^{-16}\text{pt} = 1.52587890625e-5\text{pt}.$$

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` ⋆	`\fp_abs:n {⟨floating point expression⟩}`
New: 2012-05-14	
Updated: 2012-07-08	

Evaluates the ⟨*floating point expression*⟩ as described for `\fp_eval:n` and leaves the absolute value of the result in the input stream. This function does not raise any exception beyond those raised when evaluating its argument. Within floating point expressions, `abs()` can be used.

`\fp_max:nn {⟨fp expression 1⟩} {⟨fp expression 2⟩}`

Evaluates the ⟨*floating point expressions*⟩ as described for `\fp_eval:n` and leaves the resulting larger (`max`) or smaller (`min`) value in the input stream. This function does not raise any exception beyond those raised when evaluating its argument. Within floating point expressions, `max()` and `min()` can be used.

10 Disclaimer and roadmap

The package may break down if the escape character is among `0123456789_+`; 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.

- Decide what exponent range to consider.

- Support signalling `nan`.

- Modulo and remainder, and rounding functions `quantize`, `quantize0`, `quantize+`, `quantize-`, `quantize=`, `round=`. Should the modulo also be provided as (catcode 12) `%`?

- `\fp_format:nn {⟨fpexpr⟩} {⟨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`.

- Random numbers (pgfmath provides `rnd`, `rand`, `random`), with seed reset at every `\fp_set:Nn`.

- 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 `\fp_if_nan:nTF`, and an `isnan` function?

- Support keyword arguments?

Pgfmath also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs. (Exclamation points mark important bugs.)

- 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.

- The overflow trap receives the wrong argument in l3fp-expo (see `exp(1e5678)` in `m3fp-traps001`).

  Possible optimizations/improvements.

- Document that l3trial/l3fp-types introduces tools for adding new types.

- In subsection 9.1, write a grammar.

- Fix the `TWO BARS` business with the index.

- 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)? Perhaps for including comments inside the computation itself??

Part XXIII

The **l3candidates** package
Experimental additions to **l3kernel**

1 Important notice

This module provides a space in which functions can be added to l3kernel (expl3) while still being experimental.

> **As such, the functions here may not remain in their current form, or indeed at all, in l3kernel in the future.**

In contrast to the material in l3experimental, the functions here are all *small* additions to the kernel. We encourage programmers to test them out and report back on the `LaTeX-L` mailing list.

Thus, if you intend to use any of these functions from the candidate module in a public package offered to others for productive use (e.g., being placed on CTAN) please consider the following points carefully:

- Be prepared that your public packages might require updating when such functions are being finalized.

- Consider informing us that you use a particular function in your public package, e.g., by discussing this on the `LaTeX-L` mailing list. This way it becomes easier to coordinate any updates necessary without issues for the users of your package.

- Discussing and understanding use cases for a particular addition or concept also helps to ensure that we provide the right interfaces in the final version so please give us feedback if you consider a certain candidate function useful (or not).

We only add functions in this space if we consider them being serious candidates for a final inclusion into the kernel. However, real use sometimes leads to better ideas, so functions from this module are **not necessarily stable** and we may have to adjust them!

2 Additions to **l3basics**

\cs_log:N
\cs_log:c

New: 2014-08-22
Updated: 2015-08-03

\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.

__kernel_register_log:N
__kernel_register_log:c

Updated: 2015-08-03

__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.

3 Additions to **l3box**

3.1 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_resize:Nnn
\box_resize:cnn

\box_resize:Nnn ⟨box⟩ {⟨x-size⟩} {⟨y-size⟩}

Resize the ⟨box⟩ to ⟨x-size⟩ horizontally and ⟨y-size⟩ vertically (both of the sizes are dimension expressions). The ⟨y-size⟩ is the vertical size (height plus depth) of the box. The updated ⟨box⟩ will be an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. Negative sizes will cause the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ will be unchanged. Thus negative y-sizes will result in a box a depth dependent on the height of the original box a height dependent on the depth. The resizing applies within the current TeX group level.

\box_resize_to_ht_plus_dp:Nn
\box_resize_to_ht_plus_dp:cn

\box_resize_to_ht_plus_dp:Nn ⟨box⟩ {⟨y-size⟩}

Resize 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 vertical size (height plus depth) of the box. The updated ⟨box⟩ will be an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. A negative size will cause the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ will be unchanged. Thus negative y-sizes will result in a box with depth dependent on the height of the original box and height dependent on the depth of the original. The resizing applies within the current TeX group level.

\box_resize_to_ht:Nn
\box_resize_to_ht:cn

\box_resize_to_ht:Nn ⟨box⟩ {⟨y-size⟩}

Resize 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, not including depth, of the box. The updated ⟨box⟩ will be an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. A negative size will cause the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ will be unchanged. Thus negative y-sizes will result in a box with depth dependent on the height of the original box and height dependent on the depth of the original. The resizing applies within the current TeX group level.

\box_resize_to_wd:Nn
\box_resize_to_wd:cn

\box_resize_to_wd:Nn ⟨box⟩ {⟨x-size⟩}

Resize the ⟨box⟩ to ⟨x-size⟩ horizontally, scaling the vertical size by the same amount (⟨x-size⟩ is a dimension expression). The updated ⟨box⟩ will be an hbox, irrespective of the nature of the ⟨box⟩ before the resizing is applied. A negative size will cause the material in the ⟨box⟩ to be reversed in direction, but the reference point of the ⟨box⟩ will be unchanged. Thus negative y-sizes will result in a box a depth dependent on the height of the original box a height dependent on the depth. The resizing applies within the current TeX group level.

`\box_resize_to_wd_and_ht:Nnn`
`\box_resize_to_wd_and_ht:cnn`

New: 2014-07-03

`\box_resize_to_wd_and_ht:Nnn` ⟨*box*⟩ {⟨*x-size*⟩} {⟨*y-size*⟩}

Resize the ⟨*box*⟩ to a *height* of ⟨*x-size*⟩ horizontally and ⟨*y-size*⟩ vertically (both of the sizes are dimension expressions). The ⟨*y-size*⟩ is the *height* of the box, ignoring any depth. The updated ⟨*box*⟩ will be an hbox, irrespective of the nature of the ⟨*box*⟩ before the resizing is applied. Negative sizes will cause the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ will be unchanged.

`\box_rotate:Nn`
`\box_rotate:cn`

`\box_rotate:Nn` ⟨*box*⟩ {⟨*angle*⟩}

Rotates the ⟨*box*⟩ by ⟨*angle*⟩ (in degrees) anti-clockwise about its reference point. The reference point of the updated box will be moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated ⟨*box*⟩ will be an hbox, irrespective of the nature of the ⟨*box*⟩ before the rotation is applied. The rotation applies within the current TeX group level.

`\box_scale:Nnn`
`\box_scale:cnn`

`\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 integer expressions). The updated ⟨*box*⟩ will be an hbox, irrespective of the nature of the ⟨*box*⟩ before the scaling is applied. Negative scalings will cause the material in the ⟨*box*⟩ to be reversed in direction, but the reference point of the ⟨*box*⟩ will be unchanged. Thus negative *y*-scales will result in a box a depth dependent on the height of the original box a height dependent on the depth. The resizing applies within the current TeX group level.

3.2 Viewing part of a box

`\box_clip:N`
`\box_clip:c`

`\box_clip:N` ⟨*box*⟩

Clips the ⟨*box*⟩ in the output so that only material inside the bounding box is displayed in the output. The updated ⟨*box*⟩ will be an hbox, irrespective of the nature of the ⟨*box*⟩ before the clipping is applied. The clipping applies within the current TeX group level.

These functions require the LaTeX3 native drivers: they will not work with the LaTeX 2ε graphics drivers!

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_trim:Nnnnn`
`\box_trim:cnnnn`

`\box_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 fourth. All adjustments are ⟨*dimension expressions*⟩. Material output of the bounding box will still be displayed in the output unless `\box_clip:N` is subsequently applied. The updated ⟨*box*⟩ will be an hbox, irrespective of the nature of the ⟨*box*⟩ before the trim operation is applied. The adjustment applies within the current TeX group level. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

`\box_viewport:Nnnnn`
`\box_viewport:cnnnn`

`\box_viewport:Nnnnn ⟨box⟩ {⟨llx⟩} {⟨lly⟩} {⟨urx⟩} {⟨ury⟩}`

Adjusts the bounding box of the ⟨*box*⟩ such that it has lower-left co-ordinates (⟨*llx*⟩, ⟨*lly*⟩) and upper-right co-ordinates (⟨*urx*⟩, ⟨*ury*⟩). All four co-ordinate positions are ⟨*dimension expressions*⟩. Material output of the bounding box will still be displayed in the output unless `\box_clip:N` is subsequently applied. The updated ⟨*box*⟩ will be an hbox, irrespective of the nature of the ⟨*box*⟩ before the viewport operation is applied. The adjustment applies within the current TeX group level.

4 Additions to **l3clist**

`\clist_log:N`
`\clist_log:c`
New: 2014-08-22

`\clist_log:N ⟨comma list⟩`

Writes the entries in the ⟨*comma list*⟩ 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.

5 Additions to **l3coffins**

`\coffin_resize:Nnn`
`\coffin_resize:cnn`

`\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_rotate:Nn ⟨coffin⟩ {⟨angle⟩}`

Rotates the ⟨*coffin*⟩ by the given ⟨*angle*⟩ (given in degrees counter-clockwise). This process will rotate both the coffin content and poles. Multiple rotations will not result in the bounding box of the coffin growing unnecessarily.

`\coffin_scale:Nnn`
`\coffin_scale:cnn`

`\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.

`\coffin_log_structure:N`
`\coffin_log_structure:c`
New: 2014-08-22

`\coffin_log_structure:N ⟨coffin⟩`

This function writes the structural information about the ⟨*coffin*⟩ in the log file. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin. See also `\coffin_show_structure:N` which displays the result in the terminal.

6 Additions to **l3file**

`\file_if_exist_input:n {⟨file name⟩}`
`\file_if_exist_input:nTF {⟨file name⟩} {⟨true code⟩} {⟨false code⟩}`

Searches for ⟨*file name*⟩ using the current TEX search path and the additional paths controlled by `\file_path_include:n`). If found, inserts the ⟨*true code*⟩ then reads in the file as additional LATEX source as described for `\file_input:n`. Note that `\file_-if_exist_input:n` does not raise an error if the file is not found, in contrast to `\file_-input:n`.

`\ior_map_inline:Nn ⟨stream⟩ {⟨inline function⟩}`

Applies the ⟨*inline function*⟩ to ⟨*lines*⟩ obtained by reading one or more lines (until an equal number of left and right braces are found) from the ⟨*stream*⟩. The ⟨*inline function*⟩ should consist of code which will receive 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_str_map_inline:Nn {⟨stream⟩} {⟨inline function⟩}`

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 will receive 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_break:`

Used to terminate a `\ior_map_...` function before all lines from the ⟨*stream*⟩ have been processed. This will normally take 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 will lead to low level TEX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro `\__prg_break_point:Nn` before further items are taken from the input stream. This will depend on the design of the mapping function.

\ior_map_break:n   \ior_map_break:n {⟨tokens⟩}

New: 2012-06-29

Used to terminate a \ior_map_... function before all lines in the ⟨*stream*⟩ have been processed, inserting the ⟨*tokens*⟩ after the mapping has ended. This will normally take place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_ior
  {
    \str_if_eq:nnTF { #1 } { bingo }
      { \ior_map_break:n { <tokens> } }
      {
        % Do something useful
      }
  }
```

Use outside of a \ior_map_... scenario will lead to low level TEX errors.

TEXhackers note: When the mapping is broken, additional tokens may be inserted by the internal macro __prg_break_point:Nn before the ⟨*tokens*⟩ are inserted into the input stream. This will depend on the design of the mapping function.

\ior_log_streams: \ior_log_streams:
\iow_log_streams: \iow_log_streams:

New: 2014-08-22

Writes in the log file a list of the file names associated with each open stream: intended for tracking down problems.

7 Additions to **l3fp**

\fp_log:N   \fp_log:N ⟨fp var⟩
\fp_log:c   \fp_log:n {⟨floating point expression⟩}
\fp_log:n

New: 2014-08-22
Updated: 2015-08-07

Evaluates the ⟨*floating point expression*⟩ and writes the result in the log file.

8 Additions to **l3int**

\int_log:N   \int_log:N ⟨integer⟩
\int_log:c

New: 2014-08-22
Updated: 2015-08-03

Writes the value of the ⟨*integer*⟩ in the log file.

\int_log:n   \int_log:n {⟨integer expression⟩}

New: 2014-08-22
Updated: 2015-08-07

Writes the result of evaluating the ⟨*integer expression*⟩ in the log file.

9 Additions to **l3keys**

`\keys_log:nn` `\keys_log:nn {⟨module⟩} {⟨key⟩}`

Writes in the log file the function which is used to actually implement a ⟨*key*⟩ for a ⟨*module*⟩.

10 Additions to **l3msg**

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, the message text and arguments are not expanded, 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. 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` ⋆

`\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 `\::error` 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 Additions to **l3prg**

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 } } % is skipped
```

```
      }
   }
   { ! \int_compare_p:n { 2 = 4 } }
```

the line marked with `is 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_lazy_all_p:n` ⋆
`\bool_lazy_all:nTF` ⋆

New: 2015-11-15

`\bool_lazy_all_p:n { {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} ⋯ {⟨boolexpr_N⟩} }`
`\bool_lazy_all:nTF { {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} ⋯ {⟨boolexpr_N⟩} } {⟨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`
will be 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

`\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 ⟨*boolexpr₂*⟩ will only be 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

`\bool_lazy_any_p:n { {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} ⋯ {⟨boolexpr_N⟩} }`
`\bool_lazy_any:nTF { {⟨boolexpr₁⟩} {⟨boolexpr₂⟩} ⋯ {⟨boolexpr_N⟩} } {⟨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` will be 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

`\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 ⟨*boolexpr₂*⟩ will only be 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_log:N`
`\bool_log:c`

New: 2014-08-22
Updated: 2015-08-03

`\bool_log:N ⟨boolean⟩`

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

`\bool_log:n`

New: 2014-08-22
Updated: 2015-08-07

`\bool_log:n {⟨boolean expression⟩}`

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

12 Additions to **l3prop**

\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_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 } }

will expand 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_log:N
\prop_log:c

New: 2014-08-12

\prop_log:N ⟨property list⟩

Writes the entries in the ⟨property list⟩ in the log file.

13 Additions to **l3seq**

\seq_mapthread_function:NNN ☆
\seq_mapthread_function:(NcN|cNN|ccN) ☆

\seq_mapthread_function:NNN ⟨seq₁⟩ ⟨seq₂⟩ ⟨function⟩

Applies ⟨function⟩ to every pair of items ⟨seq₁-item⟩–⟨seq₂-item⟩ from the two sequences, returning items from both sequences from left to right. The ⟨function⟩ will receive two n-type arguments for each iteration. The mapping will terminate when the end of either sequence is reached (*i.e.* whichever sequence has fewer items determines how many iterations occur).

\seq_set_filter:NNn
\seq_gset_filter:NNn

\seq_set_filter:NNn ⟨sequence₁⟩ ⟨sequence₂⟩ {⟨inline boolexpr⟩}

Evaluates the ⟨inline boolexpr⟩ for every ⟨item⟩ stored within the ⟨sequence₂⟩. The ⟨inline boolexpr⟩ will receive the ⟨item⟩ as #1. The sequence of all ⟨items⟩ for which the ⟨inline boolexpr⟩ evaluated to true is assigned to ⟨sequence₁⟩.

TEXhackers note: Contrarily to other mapping functions, \seq_map_break: cannot be used in this function, and will lead to low-level TEX errors.

\seq_set_map:NNn
\seq_gset_map:NNn

New: 2011-12-22

`\seq_set_map:NNn` ⟨sequence₁⟩ ⟨sequence₂⟩ {⟨inline function⟩}

Applies ⟨inline function⟩ to every ⟨item⟩ stored within the ⟨sequence₂⟩. The ⟨inline function⟩ should consist of code which will receive the ⟨item⟩ as #1. The sequence resulting from x-expanding ⟨inline function⟩ applied to each ⟨item⟩ is assigned to ⟨sequence₁⟩. 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 will lead to low-level TEX errors.

\seq_log:N
\seq_log:c

New: 2014-08-12

`\seq_log:N` ⟨sequence⟩

Writes the entries in the ⟨sequence⟩ in the log file.

14 Additions to **l3skip**

\skip_split_finite_else_action:nnNN

`\skip_split_finite_else_action:nnNN` {⟨skipexpr⟩} {⟨action⟩}
                                        ⟨dimen₁⟩ ⟨dimen₂⟩

Checks if the ⟨skipexpr⟩ contains finite glue. If it does then it assigns ⟨dimen₁⟩ the stretch component and ⟨dimen₂⟩ the shrink component. If it contains infinite glue set ⟨dimen₁⟩ and ⟨dimen₂⟩ to 0 pt and place #2 into the input stream: this is usually an error or warning message of some sort.

\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

New: 2014-08-22
Updated: 2015-08-07

`\dim_log:n` {⟨dimension expression⟩}

Writes the result of evaluating the ⟨dimension expression⟩ in the log file.

\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 expression⟩}

Writes the result of evaluating the ⟨skip expression⟩ in the log file.

\muskip_log:N
\muskip_log:c

New: 2014-08-22
Updated: 2015-08-03

`\muskip_log:N` ⟨muskip⟩

Writes the value of the ⟨muskip⟩ in the log file.

\muskip_log:n   \muskip_log:n {⟨muskip expression⟩}

New: 2014-08-22   Writes the result of evaluating the ⟨muskip expression⟩ in the log file.
Updated: 2015-08-07

15 Additions to **l3tl**

\tl_if_single_token_p:n ⋆   \tl_if_single_token_p:n {⟨token list⟩}
\tl_if_single_token:n*TF* ⋆   \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.

\tl_reverse_tokens:n ⋆   \tl_reverse_tokens:n {⟨tokens⟩}

This function, which works directly on TeX tokens, reverses the order of the ⟨tokens⟩:
the first will be the last and the last will become first. Spaces are preserved. The reversal
also operates within brace groups, but the braces themselves are not exchanged, as this
would lead to an unbalanced token list. For instance, \tl_reverse_tokens:n {a~{b()}}
leaves {}()(b)~a in the input stream. This function requires two steps of expansion.

TeXhackers note: The result is returned within the \unexpanded primitive (\exp_not:n),
which means that the token list will not expand further when appearing in an x-type argument
expansion.

\tl_count_tokens:n ⋆   \tl_count_tokens:n {⟨tokens⟩}

Counts the number of TeX tokens in the ⟨tokens⟩ 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. This function requires three expansions, giving
an ⟨integer denotation⟩.

\tl_lower_case:n   ⋆   \tl_upper_case:n   {⟨tokens⟩}
\tl_upper_case:n   ⋆   \tl_upper_case:nn {⟨language⟩} {⟨tokens⟩}
\tl_mixed_case:n   ⋆
\tl_lower_case:nn ⋆   These functions are intended to be applied to input which may be regarded broadly
\tl_upper_case:nn ⋆   as "text". They traverse the ⟨tokens⟩ and change the case of characters as discussed
\tl_mixed_case:nn ⋆   below. The character code of the characters replaced may be arbitrary: the replacement
New: 2014-06-30   characters will have standard document-level category codes (11 for letters, 12 for letter-
Updated: 2016-01-12   like characters which can also be case-changed). Begin-group and end-group characters
                        in the ⟨tokens⟩ are normalized and become { and }, respectively.

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_lower_case:n, \str_upper_case:n and \str_fold_case:n.

The functions perform expansion on the input in most cases. In particular, input
in the form of token lists or expandable functions will be expanded *unless* it falls within
one of the special handling classes described below. This expansion approach means that
in general the result of case changing will match the "natural" outcome expected from a
"functional" approach to case modification. For example

```
\tl_set:Nn \l_tmpa_tl { hello }
\tl_upper_case:n { \l_tmpa_tl \c_space_tl world }
```

will produce

```
HELLO WORLD
```

The expansion approach taken means that in package mode any LaTeX 2ε "robust" commands which may appear in the input should be converted to engine-protected versions using for example the `\robustify` command from the etoolbox package.

\l_tl_case_change_math_tl

Case changing will not take place within math mode material so for example

```
\tl_upper_case:n { Some~text~$y = mx + c$~with~{Braces} }
```

will become

```
SOME TEXT $y = mx + c$ WITH {BRACES}
```

Material inside math mode is left entirely unchanged: in particular, no expansion is undertaken.

Detection of math mode is controlled by the list of tokens in `\l_tl_case_change_-math_tl`, which should be in open–close pairs. In package mode the standard settings is

```
$ $ \( \)
```

Note that while expansion occurs when searching the text it does not apply to math mode material (which should be unaffected by case changing). As such, whilst the opening token for math mode may be "hidden" inside a command/macro, the closing one cannot be as this is being searched for in math mode. Typically, in the types of "text" the case changing functions are intended to apply to this should not be an issue.

\l_tl_case_change_exclude_tl

Case changing can be prevented by using any command on the list `\l_tl_case_change_-exclude_tl`. Each entry should be a function to be followed by one argument: the latter will be preserved as-is with no expansion. Thus for example following

```
\tl_put_right:Nn \l_tl_case_change_exclude_tl { \NoChangeCase }
```

the input

```
\tl_upper_case:n
  { Some~text~$y = mx + c$~with~\NoChangeCase {Protection} }
```

will result in

```
SOME TEXT $y = mx + c$ WITH \NoChangeCase {Protection}
```

Notice that the case changing mapping preserves the inclusion of the escape functions: it is left to other code to provide suitable definitions (typically equivalent to `\use:n`). In particular, the result of case changing is returned protected by `\exp_not:n`.

When used with LaTeX 2ε the commands `\cite`, `\ensuremath`, `\label` and `\ref` are automatically included in the list for exclusion from case changing.

`\l_tl_case_change_accents_tl`

This list specifies accent commands which should be left unexpanded in the output. This allows for example

    \tl_upper_case:n { \" { a } }

to yield

    \" { A }

irrespective of the expandability of `\"`.

The standard contents of this variable is `\"`, `\'`, `\.`, `\^`, `` \` ``, `\~`, `\c`, `\H`, `\k`, `\r`, `\t`, `\u` and `\v`.

"Mixed" case conversion may be regarded informally as converting the first character of the ⟨*tokens*⟩ to upper case and the rest to lower case. However, the process is more complex than this as there are some situations where a single lower case character maps to a special form, for example `ij` in Dutch which becomes IJ. As such, `\tl_mixed_-case:n(n)` implement a more sophisticated mapping which accounts for this and for modifying accents on the first letter. Spaces at the start of the ⟨*tokens*⟩ are ignored when finding the first "letter" for conversion.

    \tl_mixed_case:n { hello~WORLD }   % => "Hello world"
    \tl_mixed_case:n { ~hello~WORLD } % => " Hello world"
    \tl_mixed_case:n { {hello}~WORLD } % => "{Hello} world"

When finding the first "letter" for this process, any content in math mode or covered by `\l_tl_case_change_exclude_tl` is ignored.

(Note that the Unicode Consortium describe this as "title case", but that in English title case applies on a word-by-word basis. The "mixed" case implemented here is a lower level concept needed for both "title" and "sentence" casing of text.)

`\l_tl_mixed_case_ignore_tl`

The list of characters to ignore when searching for the first "letter" in mixed-casing is determined by `\l_tl_mixed_change_ignore_tl`. This has the standard setting

    ([{ ` -

where comparisons are made on a character basis.

As is generally true for expl3, these functions are designed to work with Unicode input only. As such, UTF-8 input is assumed for *all* engines. When used with X∃TEX or LuaTEX a full range of Unicode transformations are enabled. Specifically, the standard mappings here follow those defined by the Unicode Consortium in `UnicodeData.txt` and `SpecialCasing.txt`. In the case of 8-bit engines, mappings are provided for characters which can be represented in output typeset using the `T1` font encoding. Thus for example Ãđ can be case-changed using pdfTEX. For pTEX only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Context-sensitive mappings are enabled: language-dependent cases are discussed below. Context detection will expand input but treats any unexpandable control sequences as "failures" to match a context.

Language-sensitive conversions are enabled using the ⟨*language*⟩ argument, and follow Unicode Consortium guidelines. Currently, the languages recognised for special handling are as follows.

- 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 lower casing I-dot and introduced when upper casing i-dotless.

- German (`de-alt`). An alternative mapping for German in which the lower case *Eszett* maps to a *großes Eszett*.

- Lithuanian (`lt`). The lower case letters i and j should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lower casing of the relevant upper case letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when upper casing in these cases. Note that *only* the accents used in Lithuanian are covered: the behaviour of other accents are not modified.

- Dutch (`nl`). Capitalisation of `ij` at the beginning of mixed cased input produces `IJ` rather than `Ij`. The output retains two separate letters, thus this transformation *is* available using pdfTeX.

Creating additional context-sensitive mappings requires knowledge of the underlying mapping implementation used here. The team are happy to add these to the kernel where they are well-documented (*e.g.* in Unicode Consortium or relevant government publications).

`\tl_set_from_file:Nnn`
`\tl_set_from_file:cnn`
`\tl_gset_from_file:Nnn`
`\tl_gset_from_file:cnn`

New: 2014-06-25

`\tl_set_from_file:Nnn` ⟨*tl*⟩ {⟨*setup*⟩} {⟨*filename*⟩}

Defines ⟨*tl*⟩ to the contents of ⟨*filename*⟩. Category codes may need to be set appropriately via the ⟨*setup*⟩ argument.

`\tl_set_from_file_x:Nnn`
`\tl_set_from_file_x:cnn`
`\tl_gset_from_file_x:Nnn`
`\tl_gset_from_file_x:cnn`

New: 2014-06-25

`\tl_set_from_file_x:Nnn` ⟨*tl*⟩ {⟨*setup*⟩} {⟨*filename*⟩}

Defines ⟨*tl*⟩ to the contents of ⟨*filename*⟩, expanding the contents of the file as it is read. Category codes and other definitions may need to be set appropriately via the ⟨*setup*⟩ argument.

`\tl_log:N`
`\tl_log:c`

New: 2014-08-22
Updated: 2015-08-01

`\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`

New: 2014-08-22

`\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.

16 Additions to **l3tokens**

`\peek_N_type:TF {⟨true code⟩} {⟨false code⟩}`

Tests if the next ⟨*token*⟩ in the input stream can be safely grabbed as an N-type argument. The test will be ⟨*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 will take the ⟨*false*⟩ branch, even though the next ⟨*token*⟩ is in fact a valid N-type argument. The ⟨*token*⟩ will be left in the input stream after the ⟨*true code*⟩ or ⟨*false code*⟩ (as appropriate to the result of the test).

Part XXIV

The **l3sys** package
System/runtime functions

1   The name of the job

\c_sys_jobname_str

New: 2015-09-19

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

TEXhackers note: This copies the contents of the primitive `\jobname`. It is a constant that is set by TEX and should not be overwritten by the package.

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 primitives can be altered by the user, this interface to the time and date is intended to be the "real" values.

2.1   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: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: will be one of `luatex`, `pdftex`, `ptex`, `uptex` or `xetex`.

2.2 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:TF {⟨true code⟩} {⟨false code⟩}`

Conditionals which give the current output mode the TEX run is operating in. This will always be 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: will be one of `dvi` or `pdf`.

Part XXV

The **l3luatex** package
LuaTeX-specific functions

1 Breaking out to Lua

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 or XeTeX these will raise an error: use `\sys_if_engine_luatex:T` to avoid this. Details of coding the LuaTeX engine are detailed in the LuaTeX manual.

1.1 TeX code interfaces

`\lua_now_x:n ⋆`
`\lua_now:n ⋆`

`\lua_now:n {⟨token list⟩}`

The ⟨*token list*⟩ is first tokenized by TeX, which will include 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 will execute the ⟨*Lua input*⟩ immediately, and in an expandable manner.

In the case of the `\lua_now_x:n` version the input is fully expanded by TeX in an `x`-type manner *but* the function remains fully expandable.

TeXhackers note: `\lua_now_x:n` is a macro wrapper around `\directlua`: when LuaTeX is in use two expansions will be required to yield the result of the Lua code.

`\lua_shipout_x:n`
`\lua_shipout:n`

`\lua_shipout:n {⟨token list⟩}`

The ⟨*token list*⟩ is first tokenized by TeX, which will include 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_x:n` version the input is fully expanded by TeX in an `x`-type manner during the shipout operation.

TeXhackers note: At a TeX level, the ⟨*Lua input*⟩ is stored as a "whatsit".

`\lua_escape_x:n` ⋆
`\lua_escape:n` ⋆

`\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.

In the case of the `\lua_escape_x:n` version the input is fully expanded by TEX in an `x`-type manner *but* the function remains fully expandable.

TEXhackers note: `\lua_escape_x:n` is a macro wrapper around `\luaescapestring`: when LuaTEX is in use two expansions will be required to yield the result of the Lua code.

1.2 Lua interfaces

As well as interfaces for TEX, there are a small number of Lua functions provided here. Currently these are intended for internal use only.

`l3kernel.strcmp`

`\l3kernel.strcmp(⟨str one⟩, ⟨str two⟩)`

Compares the two strings and returns `0` to TEX if the two are identical.

`l3kernel.charcat`

`\l3kernel.charcat(⟨charcode⟩, ⟨catcode⟩)`

Constructs a character of ⟨*charcode*⟩ and ⟨*catcode*⟩ and returns the result to TEX.

Part XXVI

The **l3drivers** package
Drivers

TeX relies on drivers in order to carry out a number of tasks, such as using color, including graphics and setting up hyper-links. The nature of the code required depends on the exact driver in use. Currently, LaTeX3 is aware of the following drivers:

- `pdfmode`: The "driver" for direct PDF output by *both* pdfTeX and LuaTeX (no separate driver is used in this case: the engine deals with PDF creation itself).

- `dvips`: The `dvips` program, which works in conjugation with pdfTeX or LuaTeX in DVI mode.

- `dvipdfmx`: The `dvipdfmx` program, which works in conjugation with pdfTeX or LuaTeX in DVI mode.

- `dvisvgm`: The `dvisvgm` program, which works in conjugation with pdfTeX or LuaTeX in DVI mode to create SVG output.

- `xdvipdfmx`: The driver used by X⅁TeX.

The code here is all very low-level, and should not in general be used outside of the kernel. It is also important to note that many of the functions here are closely tied to the immediate level "up", and they must be used in the correct contexts.d

1 Box clipping

__driver_box_use_clip:N

New: 2011-11-11

__driver_box_use_clip:N ⟨box⟩

Inserts the content of the ⟨box⟩ at the current insertion point such that any material outside of the bounding box will not be displayed by the driver. The material in the ⟨box⟩ is still placed in the output stream: the clipping takes place at a driver level.

This function should only be used within a surrounding horizontal box construct.

2 Box rotation and scaling

__driver_box_use_rotate:Nn

New: 2016-05-12

__driver_box_use_rotate:Nn ⟨box⟩ {⟨angle⟩}

Inserts the content of the ⟨box⟩ at the current insertion point rotated by the ⟨angle⟩ (expressed in degrees). The material is inserted with no apparent height or width, and is rotated such the the TeX reference point of the box is the center of rotation and remains the reference point after rotation. It is the responsibly of the code using this function to adjust the apparent size of the box to be correct at the TeX side.

This function should only be used within a surrounding horizontal box construct.

`\__driver_box_use_scale:Nnn`　　`\__driver_box_use_scale:Nnn` ⟨*box*⟩ {⟨*x-scale*⟩} {⟨*y-scale*⟩}

Inserts the content of the ⟨*box*⟩ at the current insertion point scale by the ⟨*x-scale*⟩ and ⟨*y-scale*⟩. The material is inserted with no apparent height or width. It is the responsibly of the code using this function to adjust the apparent size of the box to be correct at the TeX side.

This function should only be used within a surrounding horizontal box construct.

3　Color support

`\__driver_color_ensure_current:`　　`\__driver_color_ensure_current:`

Ensures that the color used to typeset material is that which was set when the material was placed in a box. This function is therefore required inside any "color safe" box to ensure that the box may be inserted in a location where the foreground color has been altered, while preserving the color used in the box.

4　Drawing

The drawing functions provided here are *highly* experimental. They are inspired heavily by the system layer of `pgf` (most have the same interface as the same functions in the latter's `\pgfsys@...` namespace). They are intended to form the basis for higher level drawing interfaces, which themselves are likely to be further abstracted for user access. Again, this model is heavily inspired by `pgf` and Ti*kz*.

These low level drawing interfaces abstract from the driver raw requirements but still require an appreciation of the concepts of PostScript/PDF/SVG graphic creation.

`\__driver_draw_begin:`　　`\__driver_draw_begin:`
`\__driver_draw_end:`　　⟨*content*⟩
　　`\__driver_draw_end:`

Defines a drawing environment. This will be a scope for the purposes of the graphics state. Depending on the driver, other set up may or may not take place here. The natural size of the ⟨*content*⟩ should be zero from the TeX perspective: allowance for the size of the content must be made at a higher level (or indeed this can be skipped if the content is to overlap other material).

`\__driver_draw_scope_begin:`　　`\__driver_draw_scope_begin:`
`\__driver_draw_scope_end:`　　⟨*content*⟩
　　`\__driver_draw_scope_end:`

Defines a scope for drawing settings and so on. Changes to the graphic state and concepts such as color or linewidth are localised to a scope. This function pair must never be used if an partial path is under construction: such paths must be entirely contained at one unbroken scope level. Note that scopes do not form TeX groups and may not be aligned with them.

4.1 Path construction

__driver_draw_moveto:nn

__driver_draw_move:nn {⟨x⟩} {⟨y⟩}

Moves the current drawing reference point to (⟨x⟩, ⟨y⟩); any active transformation matrix will apply.

__driver_draw_lineto:nn

__driver_draw_lineto:nn {⟨x⟩} {⟨y⟩}

Adds a path from the current drawing reference point to (⟨x⟩, ⟨y⟩); any active transformation matrix will apply. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

__driver_draw_curveto:nnnnnn

__driver_draw_curveto:nnnnnn {⟨x₁⟩} {⟨y₁⟩}
{⟨x₂⟩} {⟨y₂⟩} {⟨x₃⟩} {⟨y₃⟩}

Adds a Bezier curve path from the current drawing reference point to (⟨x₃⟩, ⟨y₃⟩), using (⟨x₁⟩, ⟨y₁⟩) and (⟨x₂⟩, ⟨y₂⟩) as control points; any active transformation matrix will apply. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

__driver_draw_rectangle:nnnn

__driver_draw_rectangle:nnnn {⟨x⟩} {⟨y⟩} {⟨width⟩} {⟨height⟩}

Adds rectangular path from (⟨x₁⟩, ⟨y₁⟩) of ⟨height⟩ and ⟨width⟩; any active transformation matrix will apply. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

__driver_draw_closepath:

__driver_draw_closepath:

Closes an existing path, adding a line from the current point to the start of path. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

4.2 Stroking and filling

__driver_draw_stroke:
__driver_draw_closestroke:

⟨path construction⟩
__driver_draw_stroke:

Draws a line alone the current path, which will also be closed when __driver_draw_-closestroke: is used. The nature of the line drawn is influenced by settings for

- Line thickness

- Stroke color (or the current color if no specific stroke color is set)

- Line capping (how non-closed line ends should look)

- Join style (how a bend in the path should be rendered)

- Dash pattern

The path may also be used for clipping.

`\__driver_draw_fill:`	⟨*path construction*⟩
`\__driver_draw_fillstroke:`	`\__driver_draw_fill:`

Fills the area surrounded by the current path: this will be closed prior to filling if it is not already. The `fillstroke` version will also stroke the path as described for `\__driver_-draw_stroke:`. The fill is influenced by the setting for fill color (or the current color if no specific stroke color is set). The path may also be used for clipping. For paths which are self-intersecting or comprising multiple parts, the determination of which areas are inside the path is made using the non-zero winding number rule unless the even-odd rule is active.

`\__driver_draw_nonzero_rule:`	`\__driver_draw_nonzero_rule:`
`\__driver_draw_evenodd_rule:`	

Active either the non-zero winding number or the even-odd rule, respectively, for determining what is inside a fill or clip area. For technical reasons, these command are not influenced by scoping and apply on an ongoing basis.

`\__driver_draw_clip:`	⟨*path construction*⟩
	`\__driver_draw_clip:`

Indicates that the current path should be used for clipping, such that any subsequent material outside of the path (but within the current scope) will not be shown. This command should be given once a path is complete but before it is stroked or filled (if appropriate). This command is *not* affected by scoping: it applies to exactly one path as shown.

`\__driver_draw_discardpath:`	⟨*path construction*⟩
	`\__driver_draw_discardpath:`

Discards the current path without stroking or filling. This is primarily useful for paths constructed purely for clipping, as this alone does not end the paths existence.

4.3 Stroke options

`\__driver_draw_linewidth:n`	`\__driver_draw_linewidth:n {`⟨*dimexpr*⟩`}`

Sets the width to be used for stroking to ⟨*dimexpr*⟩.

`\__driver_draw_dash:nn`	`\__driver_draw_dash:nn {`⟨*dash pattern*⟩`} {`⟨*phase*⟩`}`

Sets the pattern of dashing to be used when stroking a line. The ⟨*dash pattern*⟩ should be a comma-separated list of dimension expressions. This is then interpreted as a series of pairs of line-on and line-off lengths. For example `3pt, 4pt` means that 3 pt on, 4 pt off, 3 pt on, and so on. A more complex pattern will also repeat: `3pt, 4pt, 1pt, 2pt` results in 3 pt on, 4 pt off, 1 pt on, 2 pt off, 3 pt on, and so on. An odd number of entries means that the last is repeated, for example `3pt` is equal to `3pt, 3pt`. An empty pattern yields a solid line.

The ⟨*phase*⟩ specifies an offset at the start of the cycle. For example, with a pattern `3pt` a phase of `1pt` will mean that the output is 2 pt on, 3 pt off, 3 pt on, 3 pt on, *etc.*

```
\__driver_draw_cap_butt:          \__driver_draw_cap_butt:
\__driver_draw_cap_rectangle:
\__driver_draw_cap_round:
```

Sets the style of terminal stroke position to one of butt, rectangle or round.

```
\__driver_draw_join_bevel:        \__driver_draw_cap_butt:
\__driver_draw_join_miter:
\__driver_draw_join_round:
```
Sets the style of stroke joins to one of bevel, miter or round.

```
\__driver_draw_miterlimit:n       \__driver_draw_miterlimit:n {⟨dimexpr⟩}
```

Sets the miter limit of lines joined as a miter, as described in the PDF and PostScript manuals.

4.4 Color

```
\__driver_draw_color_cmyk:nnnn       \__driver_draw_color_cmyk:nnnn {⟨cyan⟩} {⟨magneta⟩} {⟨yellow⟩}
\__driver_draw_color_cmyk_fill:nnnn  {⟨black⟩}
\__driver_draw_color_cmyk_stroke:nnnn
```

Sets the color for drawing to the CMYK values specified, all of which are fp expressions which should evaluate to between 0 and 1. The fill and stroke versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

```
\__driver_draw_color_gray:n         \__driver_draw_color_gray:n {⟨gray⟩}
\__driver_draw_color_gray_fill:n
\__driver_draw_color_gray_stroke:n
```

Sets the color for drawing to the grayscale value specified, which is fp expressions which should evaluate to between 0 and 1. The fill and stroke versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

```
\__driver_draw_color_rgb:nnn        \__driver_draw_color_gray:n {⟨red⟩} {⟨green⟩} {⟨blue⟩}
\__driver_draw_color_rgb_fill:nnn
\__driver_draw_color_rgb_stroke:nnn
```

Sets the color for drawing to the RGB values specified, all of which are fp expressions which should evaluate to between 0 and 1. The fill and stroke versions set only the color for those operations. Note that the general setting is more efficient with some drivers so should in most cases be preferred.

4.5 Inserting TeX material

$\__driver\_draw\_hbox:Nnnnnnn$ $\langle box \rangle$
{$\langle a \rangle$} {$\langle b \rangle$} {$\langle c \rangle$} {$\langle d \rangle$} {$\langle x \rangle$} {$\langle y \rangle$}

Inserts the ⟨*box*⟩ as an hbox with the box reference point placed at (*x*, *y*). The transformation matrix [*abcd*] will be applied to the box, allowing it to be in synchronisation with any scaling, rotation or skewing applying more generally. Note that TeX material should not be inserted directly into a drawing as it will not be in the correct location. Also note that as for other drawing elements the box here will have no size from a TeX perspective.

4.6 Coordinate system transformations

$\__driver\_draw\_transformcm:nnnnnn$ {$\langle a \rangle$} {$\langle b \rangle$} {$\langle c \rangle$} {$\langle d \rangle$}
{$\langle x \rangle$} {$\langle y \rangle$}

Applies the transformation matrix [*abcd*] and offset vector (*x*, *y*) to the current graphic state. This will affect any subsequent items in the same scope but not those already given.

Part XXVII
Implementation

1 **l3bootstrap** implementation

₁ ⟨*initex | package⟩
₂ ⟨@@=expl⟩

1.1 Format-specific code

The very first thing to do is to bootstrap the iniTeX system so that everything else will actually work. TeX does not start with some pretty basic character codes set up.

₃ ⟨*initex⟩
₄ \catcode '\{ = 1 %
₅ \catcode '\} = 2 %
₆ \catcode '\# = 6 %
₇ \catcode '\^ = 7 %
₈ ⟨/initex⟩

Tab characters should not show up in the code, but to be on the safe side.

₉ ⟨*initex⟩
₁₀ \catcode '\^^I = 10 %
₁₁ ⟨/initex⟩

For LuaTeX, the extra primitives need to be enabled. This is not needed in package mode: common formats have the primitives enabled.

₁₂ ⟨*initex⟩
₁₃ \begingroup\expandafter\expandafter\expandafter\endgroup
₁₄ \expandafter\ifx\csname directlua\endcsname\relax
₁₅ \else

```
16    \directlua{tex.enableprimitives("", tex.extraprimitives())}%
17  \fi
18  ⟨/initex⟩
```

Depending on the versions available, the LaTeX format may not have the raw \Umath primitive names available. We fix that globally: it should cause no issues. Older LuaTeX versions do not have a pre-built table of the primitive names here so sort one out ourselves. These end up globally-defined but at that is true with a newer format anyway and as they all start \U this should be reasonably safe.

```
19  ⟨*package⟩
20  \begingroup
21    \expandafter\ifx\csname directlua\endcsname\relax
22    \else
23      \directlua{%
24        local i
25        local t = { }
26        for _,i in pairs(tex.extraprimitives("luatex")) do
27          if string.match(i,"^U") then
28            if not string.match(i,"^Uchar$") then
29              table.insert(t,i)
30            end
31          end
32        end
33        tex.enableprimitives("", t)
34      }%
35    \fi
36  \endgroup
37  ⟨/package⟩
```

1.2 The \pdfstrcmp primitive in XꟲTEX

Only pdfTEX has a primitive called \pdfstrcmp. The XꟲTEX version is just \strcmp, so there is some shuffling to do. As this is still a real primitive, using the pdfTEX name is "safe".

```
38  \begingroup\expandafter\expandafter\expandafter\endgroup
39    \expandafter\ifx\csname pdfstrcmp\endcsname\relax
40    \let\pdfstrcmp\strcmp
41  \fi
```

1.3 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.

```
42  \begingroup\expandafter\expandafter\expandafter\endgroup
43  \expandafter\ifx\csname directlua\endcsname\relax
44  \else
45    \ifnum\luatexversion<70 %
46    \else
```

In package mode a category code table is needed: either use a pre-loaded allocator or provide one using the LaTeX 2ε-based generic code. In format mode the table used here can be hard-coded into the Lua.

```
47  ⟨*package⟩
48      \begingroup\expandafter\expandafter\expandafter\endgroup
49      \expandafter\ifx\csname newcatcodetable\endcsname\relax
50        \input{ltluatex}%
51      \fi
52      \newcatcodetable\ucharcat@table
53      \directlua{
54        l3kernel = l3kernel or { }
55        local charcat_table = \number\ucharcat@table\space
56        l3kernel.charcat_table = charcat_table
57      }%
58  ⟨/package⟩
59      \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 will reveal what mode is in operation.

```
60      \ifnum 0%
61        \directlua{
62          if status.ini_version then
63            tex.write("1")
64          end
65        }>0 %
66        \everyjob\expandafter{%
67          \the\expandafter\everyjob
68          \csname\detokenize{lua_now_x:n}\endcsname{require("expl3")}%
69        }%
70      \fi
71    \fi
72  \fi
```

1.4   Engine requirements

The code currently requires ε-TeX and functionality equivalent to `\pdfstrcmp`, and also driver and Unicode character support. This is available in a reasonably-wide range of engines.

```
73  \begingroup
74    \def\next{\endgroup}%
75    \def\ShortText{Required primitives not found}%
76    \def\LongText%
77      {%
78        LaTeX3 requires the e-TeX primitives and additional functionality as
79        described in the README file.
80        \LineBreak
81        These are available in the engines\LineBreak
82        - pdfTeX v1.40\LineBreak
83        - XeTeX v0.9994\LineBreak
84        - LuaTeX v0.70\LineBreak
85        - e-(u)pTeX mid-2012\LineBreak
86        or later.\LineBreak
87        \LineBreak
88      }%
89    \ifnum0%
90      \expandafter\ifx\csname pdfstrcmp\endcsname\relax
```

```
 91       \else
 92         \expandafter\ifx\csname pdftexversion\endcsname\relax
 93           1%
 94         \else
 95           \ifnum\pdftexversion<140 \else 1\fi
 96         \fi
 97       \fi
 98       \expandafter\ifx\csname directlua\endcsname\relax
 99       \else
100         \ifnum\luatexversion<40 \else 1\fi
101       \fi
102       =0 %
103         \newlinechar`\^^J %
```
⟨*initex⟩
```
105         \def\LineBreak{^^J}%
106         \edef\next
107           {%
108             \errhelp
109               {%
110                 \LongText
111                 For pdfTeX and XeTeX the '-etex' command-line switch is also
112                 needed.\LineBreak
113                 \LineBreak
114                 Format building will abort!\LineBreak
115               }%
116             \errmessage{\ShortText}%
117             \endgroup
118             \noexpand\end
119           }%
```
⟨/initex⟩
⟨*package⟩
```
122         \def\LineBreak{\noexpand\MessageBreak}%
123         \expandafter\ifx\csname PackageError\endcsname\relax
124           \def\LineBreak{^^J}%
125           \def\PackageError#1#2#3%
126             {%
127               \errhelp{#3}%
128               \errmessage{#1 Error: #2}%
129             }%
130         \fi
131         \edef\next
132           {%
133             \noexpand\PackageError{expl3}{\ShortText}
134               {\LongText Loading of expl3 will abort!}%
135             \endgroup
136             \noexpand\endinput
137           }%
```
⟨/package⟩
```
139     \fi
140 \next
```

225

1.5 Extending allocators

In format mode, allocating registers is handled by l3alloc. However, in package mode it's much safer to rely on more general code. For example, the ability to extend TeX's allocation routine to allow for ε-TeX has been around since 1997 in the etex package.

Loading this support is delayed until here as we are now sure that the ε-TeX extensions and `\pdfstrcmp` or equivalent are available. Thus there is no danger of an "uncontrolled" error if the engine requirements are not met.

For LaTeX2_ε we need to make sure that the extended pool is being used: expl3 uses a lot of registers. For formats from 2015 onward there is nothing to do as this is automatic. For older formats, the etex package needs to be loaded to do the job. In that case, some inserts are reserved also as these have to be from the standard pool. Note that `\reserveinserts` is `\outer` and so is accessed here by csname. In earlier versions, loading etex was done directly and so `\reserveinserts` appeared in the code: this then required a `\relax` after `\RequirePackage` to prevent an error with "unsafe" definitions as seen for example with capoptions. The optional loading here is done using a group and `\ifx` test as we are not quite in the position to have a single name for `\pdfstrcmp` just yet.

```
141 ⟨*package⟩
142 \begingroup
143   \def\@tempa{LaTeX2e}%
144   \def\next{}%
145   \ifx\fmtname\@tempa
146     \expandafter\ifx\csname extrafloats\endcsname\relax
147       \def\next
148         {%
149           \RequirePackage{etex}%
150           \csname reserveinserts\endcsname{32}%
151         }%
152     \fi
153   \fi
154 \expandafter\endgroup
155 \next
156 ⟨/package⟩
```

1.6 Character data

TeX needs various pieces of data to be set about characters, in particular which ones to treat as letters and which `\lccode` values apply as these affect hyphenation. It makes most sense to set this and related information up in one place. Whilst for LuaTeX hyphenation patterns can be read anywhere, other engines have to build them into the format and so we *must* do this set up before reading the patterns. For the Unicode engines, there are shared loaders available to obtain the relevant information directly from the Unicode Consortium data files. These need standard (Ini)TeX category codes and primitive availability and must therefore loaded *very* early. This has a knock-on effect on the 8-bit set up: it makes sense to do the definitions for those here as well so it is all in one place.

For XeTeX and LuaTeX, which are natively Unicode engines, simply load the Unicode data.

```
157 ⟨*initex⟩
158 \ifdefined\Umathcode
```

```
159    \input load-unicode-data %
160    \input load-unicode-math-classes %
161  \else
```

For the 8-bit engines a font encoding scheme must be chosen. At present, this is the EC (`T1`) scheme, with the assumption that languages for which this is not appropriate will be used with one of the Unicode engines.

```
162    \begingroup
```

Lower case chars: map to themselves when lower casing and down by `"20` when upper casing. (The characters `a`–`z` are set up correctly by IniTeX.)

```
163    \def\temp{%
164      \ifnum\count0>\count2 %
165      \else
166        \global\lccode\count0 = \count0 %
167        \global\uccode\count0 = \numexpr\count0 - "20\relax
168        \advance\count0 by 1 %
169        \expandafter\temp
170      \fi
171    }
172    \count0 = "A0 %
173    \count2 = "BC %
174    \temp
175    \count0 = "E0 %
176    \count2 = "FF %
177    \temp
```

Upper case chars: map up by `"20` when lower casing, to themselves when upper casing and require an `\sfcode` of 999. (The characters `A`–`Z` are set up correctly by IniTeX.)

```
178    \def\temp{%
179      \ifnum\count0>\count2 %
180      \else
181        \global\lccode\count0 = \numexpr\count0 + "20\relax
182        \global\uccode\count0 = \count0 %
183        \global\sfcode\count0 = 999 %
184        \advance\count0 by 1 %
185        \expandafter\temp
186      \fi
187    }
188    \count0 = "80 %
189    \count2 = "9C %
190    \temp
191    \count0 = "C0 %
192    \count2 = "DF %
193    \temp
```

A few special cases where things are not as one might expect using the above pattern: dotless-I, dotless-J, dotted-I and d-bar.

```
194    \global\lccode'\^^Y = '\^^Y %
195    \global\uccode'\^^Y = '\I %
196    \global\lccode'\^^Z = '\^^Z %
197    \global\uccode'\^^Y = '\J %
198    \global\lccode"9D = '\i %
199    \global\uccode"9D = "9D %
200    \global\lccode"9E = "9E %
201    \global\uccode"9E = "D0 %
```

Allow hyphenation at a zero-width glyph (used to break up ligatures or to place accents between characters).

```
202      \global\lccode23 = 23 %
203    \endgroup
204  \fi
```

In all cases it makes sense to set up - to map to itself: this allows hyphenation of the rest of a word following it (suggested by Lars Helström).

```
205  \global\lccode'\- = '\- %
206  ⟨/initex⟩
```

1.7 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 will be a "do nothing" command until \ExplSyntaxOn is used. For format mode, there is no need to save category codes so that step is skipped.

```
207  \protected\def\ExplSyntaxOff{}%
208  ⟨*package⟩
209  \protected\edef\ExplSyntaxOff
210    {%
211      \protected\def\ExplSyntaxOff{}%
212      \catcode    9 = \the\catcode    9\relax
213      \catcode   32 = \the\catcode   32\relax
214      \catcode   34 = \the\catcode   34\relax
215      \catcode   38 = \the\catcode   38\relax
216      \catcode   58 = \the\catcode   58\relax
217      \catcode   94 = \the\catcode   94\relax
218      \catcode   95 = \the\catcode   95\relax
219      \catcode  124 = \the\catcode  124\relax
220      \catcode  126 = \the\catcode  126\relax
221      \endlinechar = \the\endlinechar\relax
222      \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0\relax
223    }%
224  ⟨/package⟩
```

(*End definition for* \ExplSyntaxOff. *This function is documented on page 6.*)

The code environment is now set up.

```
225  \catcode 9    = 9\relax
226  \catcode 32   = 9\relax
227  \catcode 34   = 12\relax
228  \catcode 38 =  4\relax
229  \catcode 58   = 11\relax
230  \catcode 94   = 7\relax
231  \catcode 95   = 11\relax
232  \catcode 124 = 12\relax
233  \catcode 126 = 10\relax
234  \endlinechar = 32\relax
```

\l__kernel_expl_bool The status for experimental code syntax: this is on at present.

```
235  \chardef\l__kernel_expl_bool = 1\relax
```

(*End 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` will alter the definition of `\ExplSyntaxOff` and so in package mode this function should not be used until after the end of the loading process!

```
236 \protected \def \ExplSyntaxOn
237   {
238     \bool_if:NF \l__kernel_expl_bool
239       {
240         \cs_set_protected:Npx \ExplSyntaxOff
241           {
242             \char_set_catcode:nn { 9 }   { \char_value_catcode:n { 9 } }
243             \char_set_catcode:nn { 32 }  { \char_value_catcode:n { 32 } }
244             \char_set_catcode:nn { 34 }  { \char_value_catcode:n { 34 } }
245             \char_set_catcode:nn { 38 }  { \char_value_catcode:n { 38 } }
246             \char_set_catcode:nn { 58 }  { \char_value_catcode:n { 58 } }
247             \char_set_catcode:nn { 94 }  { \char_value_catcode:n { 94 } }
248             \char_set_catcode:nn { 95 }  { \char_value_catcode:n { 95 } }
249             \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
250             \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
251             \tex_endlinechar:D =
252               \tex_the:D \tex_endlinechar:D \scan_stop:
253             \bool_set_false:N \l__kernel_expl_bool
254             \cs_set_protected:Npn \ExplSyntaxOff { }
255           }
256       }
257     \char_set_catcode_ignore:n            { 9 }   % tab
258     \char_set_catcode_ignore:n            { 32 }  % space
259     \char_set_catcode_other:n             { 34 }  % double quote
260     \char_set_catcode_alignment:n         { 38 }  % ampersand
261     \char_set_catcode_letter:n            { 58 }  % colon
262     \char_set_catcode_math_superscript:n  { 94 }  % circumflex
263     \char_set_catcode_letter:n            { 95 }  % underscore
264     \char_set_catcode_other:n             { 124 } % pipe
265     \char_set_catcode_space:n             { 126 } % tilde
266     \tex_endlinechar:D = 32 \scan_stop:
267     \bool_set_true:N \l__kernel_expl_bool
268   }
```

(*End definition for* `\ExplSyntaxOn`*. This function is documented on page 6.*)

```
269 ⟨/initex | package⟩
```

2　l3names implementation

```
270 ⟨*initex | package⟩
```

No prefix substitution here.

```
271 ⟨@@=⟩
```

The code here simply renames all of the primitives to new, internal, names. In format mode, it also deletes all of the existing names (although some do come back later).

`\tex_undefined:D`　This function does not exist at all, but is the name used by the plain TEX format for an undefined function. So it should be marked here as "taken".

(*End definition for* \tex_undefined:D.)

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.

```
272 \let \tex_global:D \global
273 \let \tex_let:D    \let
```

Everything is inside a (rather long) group, which keeps __kernel_primitive:NN trapped.

```
274 \begingroup
```

__kernel_primitive:NN A temporary function to actually do the renaming. This also allows the original names to be removed in format mode.

```
275   \long \def \__kernel_primitive:NN #1#2
276     {
277       \tex_global:D \tex_let:D #2 #1
278 ⟨*initex⟩
279       \tex_global:D \tex_let:D #1 \tex_undefined:D
280 ⟨/initex⟩
281     }
```

(*End definition for* __kernel_primitive:NN.)

To allow extracting "just the names", a bit of DocStrip fiddling.

```
282 ⟨/initex | package⟩
283 ⟨*initex | names | package⟩
```

In the current incarnation of this package, 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.

```
284   \__kernel_primitive:NN \                      \tex_space:D
285   \__kernel_primitive:NN \/                     \tex_italiccorrection:D
286   \__kernel_primitive:NN \-                     \tex_hyphen:D
```

Now all the other primitives.

```
287   \__kernel_primitive:NN \above                 \tex_above:D
288   \__kernel_primitive:NN \abovedisplayshortskip \tex_abovedisplayshortskip:D
289   \__kernel_primitive:NN \abovedisplayskip      \tex_abovedisplayskip:D
290   \__kernel_primitive:NN \abovewithdelims       \tex_abovewithdelims:D
291   \__kernel_primitive:NN \accent                \tex_accent:D
292   \__kernel_primitive:NN \adjdemerits           \tex_adjdemerits:D
293   \__kernel_primitive:NN \advance               \tex_advance:D
294   \__kernel_primitive:NN \afterassignment       \tex_afterassignment:D
295   \__kernel_primitive:NN \aftergroup            \tex_aftergroup:D
296   \__kernel_primitive:NN \atop                  \tex_atop:D
297   \__kernel_primitive:NN \atopwithdelims        \tex_atopwithdelims:D
298   \__kernel_primitive:NN \badness               \tex_badness:D
299   \__kernel_primitive:NN \baselineskip          \tex_baselineskip:D
300   \__kernel_primitive:NN \batchmode             \tex_batchmode:D
301   \__kernel_primitive:NN \begingroup            \tex_begingroup:D
302   \__kernel_primitive:NN \belowdisplayshortskip \tex_belowdisplayshortskip:D
303   \__kernel_primitive:NN \belowdisplayskip      \tex_belowdisplayskip:D
304   \__kernel_primitive:NN \binoppenalty          \tex_binoppenalty:D
305   \__kernel_primitive:NN \botmark               \tex_botmark:D
```

```
306   \__kernel_primitive:NN \box                      \tex_box:D
307   \__kernel_primitive:NN \boxmaxdepth              \tex_boxmaxdepth:D
308   \__kernel_primitive:NN \brokenpenalty            \tex_brokenpenalty:D
309   \__kernel_primitive:NN \catcode                  \tex_catcode:D
310   \__kernel_primitive:NN \char                     \tex_char:D
311   \__kernel_primitive:NN \chardef                  \tex_chardef:D
312   \__kernel_primitive:NN \cleaders                 \tex_cleaders:D
313   \__kernel_primitive:NN \closein                  \tex_closein:D
314   \__kernel_primitive:NN \closeout                 \tex_closeout:D
315   \__kernel_primitive:NN \clubpenalty              \tex_clubpenalty:D
316   \__kernel_primitive:NN \copy                     \tex_copy:D
317   \__kernel_primitive:NN \count                    \tex_count:D
318   \__kernel_primitive:NN \countdef                 \tex_countdef:D
319   \__kernel_primitive:NN \cr                       \tex_cr:D
320   \__kernel_primitive:NN \crcr                     \tex_crcr:D
321   \__kernel_primitive:NN \csname                   \tex_csname:D
322   \__kernel_primitive:NN \day                      \tex_day:D
323   \__kernel_primitive:NN \deadcycles               \tex_deadcycles:D
324   \__kernel_primitive:NN \def                      \tex_def:D
325   \__kernel_primitive:NN \defaulthyphenchar        \tex_defaulthyphenchar:D
326   \__kernel_primitive:NN \defaultskewchar          \tex_defaultskewchar:D
327   \__kernel_primitive:NN \delcode                  \tex_delcode:D
328   \__kernel_primitive:NN \delimiter                \tex_delimiter:D
329   \__kernel_primitive:NN \delimiterfactor          \tex_delimiterfactor:D
330   \__kernel_primitive:NN \delimitershortfall       \tex_delimitershortfall:D
331   \__kernel_primitive:NN \dimen                    \tex_dimen:D
332   \__kernel_primitive:NN \dimendef                 \tex_dimendef:D
333   \__kernel_primitive:NN \discretionary            \tex_discretionary:D
334   \__kernel_primitive:NN \displayindent            \tex_displayindent:D
335   \__kernel_primitive:NN \displaylimits            \tex_displaylimits:D
336   \__kernel_primitive:NN \displaystyle             \tex_displaystyle:D
337   \__kernel_primitive:NN \displaywidowpenalty      \tex_displaywidowpenalty:D
338   \__kernel_primitive:NN \displaywidth             \tex_displaywidth:D
339   \__kernel_primitive:NN \divide                   \tex_divide:D
340   \__kernel_primitive:NN \doublehyphendemerits     \tex_doublehyphendemerits:D
341   \__kernel_primitive:NN \dp                       \tex_dp:D
342   \__kernel_primitive:NN \dump                     \tex_dump:D
343   \__kernel_primitive:NN \edef                     \tex_edef:D
344   \__kernel_primitive:NN \else                     \tex_else:D
345   \__kernel_primitive:NN \emergencystretch         \tex_emergencystretch:D
346   \__kernel_primitive:NN \end                      \tex_end:D
347   \__kernel_primitive:NN \endcsname                \tex_endcsname:D
348   \__kernel_primitive:NN \endgroup                 \tex_endgroup:D
349   \__kernel_primitive:NN \endinput                 \tex_endinput:D
350   \__kernel_primitive:NN \endlinechar              \tex_endlinechar:D
351   \__kernel_primitive:NN \eqno                     \tex_eqno:D
352   \__kernel_primitive:NN \errhelp                  \tex_errhelp:D
353   \__kernel_primitive:NN \errmessage               \tex_errmessage:D
354   \__kernel_primitive:NN \errorcontextlines        \tex_errorcontextlines:D
355   \__kernel_primitive:NN \errorstopmode            \tex_errorstopmode:D
356   \__kernel_primitive:NN \escapechar               \tex_escapechar:D
357   \__kernel_primitive:NN \everycr                  \tex_everycr:D
358   \__kernel_primitive:NN \everydisplay             \tex_everydisplay:D
359   \__kernel_primitive:NN \everyhbox                \tex_everyhbox:D
```

```
360  \__kernel_primitive:NN \everyjob              \tex_everyjob:D
361  \__kernel_primitive:NN \everymath             \tex_everymath:D
362  \__kernel_primitive:NN \everypar              \tex_everypar:D
363  \__kernel_primitive:NN \everyvbox             \tex_everyvbox:D
364  \__kernel_primitive:NN \exhyphenpenalty       \tex_exhyphenpenalty:D
365  \__kernel_primitive:NN \expandafter           \tex_expandafter:D
366  \__kernel_primitive:NN \fam                   \tex_fam:D
367  \__kernel_primitive:NN \fi                    \tex_fi:D
368  \__kernel_primitive:NN \finalhyphendemerits   \tex_finalhyphendemerits:D
369  \__kernel_primitive:NN \firstmark             \tex_firstmark:D
370  \__kernel_primitive:NN \floatingpenalty       \tex_floatingpenalty:D
371  \__kernel_primitive:NN \font                  \tex_font:D
372  \__kernel_primitive:NN \fontdimen             \tex_fontdimen:D
373  \__kernel_primitive:NN \fontname              \tex_fontname:D
374  \__kernel_primitive:NN \futurelet             \tex_futurelet:D
375  \__kernel_primitive:NN \gdef                  \tex_gdef:D
376  \__kernel_primitive:NN \global                \tex_global:D
377  \__kernel_primitive:NN \globaldefs            \tex_globaldefs:D
378  \__kernel_primitive:NN \halign                \tex_halign:D
379  \__kernel_primitive:NN \hangafter             \tex_hangafter:D
380  \__kernel_primitive:NN \hangindent            \tex_hangindent:D
381  \__kernel_primitive:NN \hbadness              \tex_hbadness:D
382  \__kernel_primitive:NN \hbox                  \tex_hbox:D
383  \__kernel_primitive:NN \hfil                  \tex_hfil:D
384  \__kernel_primitive:NN \hfill                 \tex_hfill:D
385  \__kernel_primitive:NN \hfilneg               \tex_hfilneg:D
386  \__kernel_primitive:NN \hfuzz                 \tex_hfuzz:D
387  \__kernel_primitive:NN \hoffset               \tex_hoffset:D
388  \__kernel_primitive:NN \holdinginserts        \tex_holdinginserts:D
389  \__kernel_primitive:NN \hrule                 \tex_hrule:D
390  \__kernel_primitive:NN \hsize                 \tex_hsize:D
391  \__kernel_primitive:NN \hskip                 \tex_hskip:D
392  \__kernel_primitive:NN \hss                   \tex_hss:D
393  \__kernel_primitive:NN \ht                    \tex_ht:D
394  \__kernel_primitive:NN \hyphenation           \tex_hyphenation:D
395  \__kernel_primitive:NN \hyphenchar            \tex_hyphenchar:D
396  \__kernel_primitive:NN \hyphenpenalty         \tex_hyphenpenalty:D
397  \__kernel_primitive:NN \if                    \tex_if:D
398  \__kernel_primitive:NN \ifcase                \tex_ifcase:D
399  \__kernel_primitive:NN \ifcat                 \tex_ifcat:D
400  \__kernel_primitive:NN \ifdim                 \tex_ifdim:D
401  \__kernel_primitive:NN \ifeof                 \tex_ifeof:D
402  \__kernel_primitive:NN \iffalse               \tex_iffalse:D
403  \__kernel_primitive:NN \ifhbox                \tex_ifhbox:D
404  \__kernel_primitive:NN \ifhmode               \tex_ifhmode:D
405  \__kernel_primitive:NN \ifinner               \tex_ifinner:D
406  \__kernel_primitive:NN \ifmmode               \tex_ifmmode:D
407  \__kernel_primitive:NN \ifnum                 \tex_ifnum:D
408  \__kernel_primitive:NN \ifodd                 \tex_ifodd:D
409  \__kernel_primitive:NN \iftrue                \tex_iftrue:D
410  \__kernel_primitive:NN \ifvbox                \tex_ifvbox:D
411  \__kernel_primitive:NN \ifvmode               \tex_ifvmode:D
412  \__kernel_primitive:NN \ifvoid                \tex_ifvoid:D
413  \__kernel_primitive:NN \ifx                   \tex_ifx:D
```

```
414  \__kernel_primitive:NN \ignorespaces          \tex_ignorespaces:D
415  \__kernel_primitive:NN \immediate             \tex_immediate:D
416  \__kernel_primitive:NN \indent                \tex_indent:D
417  \__kernel_primitive:NN \input                 \tex_input:D
418  \__kernel_primitive:NN \inputlineno           \tex_inputlineno:D
419  \__kernel_primitive:NN \insert                \tex_insert:D
420  \__kernel_primitive:NN \insertpenalties       \tex_insertpenalties:D
421  \__kernel_primitive:NN \interlinepenalty      \tex_interlinepenalty:D
422  \__kernel_primitive:NN \jobname               \tex_jobname:D
423  \__kernel_primitive:NN \kern                  \tex_kern:D
424  \__kernel_primitive:NN \language              \tex_language:D
425  \__kernel_primitive:NN \lastbox               \tex_lastbox:D
426  \__kernel_primitive:NN \lastkern              \tex_lastkern:D
427  \__kernel_primitive:NN \lastpenalty           \tex_lastpenalty:D
428  \__kernel_primitive:NN \lastskip              \tex_lastskip:D
429  \__kernel_primitive:NN \lccode                \tex_lccode:D
430  \__kernel_primitive:NN \leaders               \tex_leaders:D
431  \__kernel_primitive:NN \left                  \tex_left:D
432  \__kernel_primitive:NN \lefthyphenmin         \tex_lefthyphenmin:D
433  \__kernel_primitive:NN \leftskip              \tex_leftskip:D
434  \__kernel_primitive:NN \leqno                 \tex_leqno:D
435  \__kernel_primitive:NN \let                   \tex_let:D
436  \__kernel_primitive:NN \limits                \tex_limits:D
437  \__kernel_primitive:NN \linepenalty           \tex_linepenalty:D
438  \__kernel_primitive:NN \lineskip              \tex_lineskip:D
439  \__kernel_primitive:NN \lineskiplimit         \tex_lineskiplimit:D
440  \__kernel_primitive:NN \long                  \tex_long:D
441  \__kernel_primitive:NN \looseness             \tex_looseness:D
442  \__kernel_primitive:NN \lower                 \tex_lower:D
443  \__kernel_primitive:NN \lowercase             \tex_lowercase:D
444  \__kernel_primitive:NN \mag                   \tex_mag:D
445  \__kernel_primitive:NN \mark                  \tex_mark:D
446  \__kernel_primitive:NN \mathaccent            \tex_mathaccent:D
447  \__kernel_primitive:NN \mathbin               \tex_mathbin:D
448  \__kernel_primitive:NN \mathchar              \tex_mathchar:D
449  \__kernel_primitive:NN \mathchardef           \tex_mathchardef:D
450  \__kernel_primitive:NN \mathchoice            \tex_mathchoice:D
451  \__kernel_primitive:NN \mathclose             \tex_mathclose:D
452  \__kernel_primitive:NN \mathcode              \tex_mathcode:D
453  \__kernel_primitive:NN \mathinner             \tex_mathinner:D
454  \__kernel_primitive:NN \mathop               \tex_mathop:D
455  \__kernel_primitive:NN \mathopen              \tex_mathopen:D
456  \__kernel_primitive:NN \mathord               \tex_mathord:D
457  \__kernel_primitive:NN \mathpunct             \tex_mathpunct:D
458  \__kernel_primitive:NN \mathrel               \tex_mathrel:D
459  \__kernel_primitive:NN \mathsurround          \tex_mathsurround:D
460  \__kernel_primitive:NN \maxdeadcycles         \tex_maxdeadcycles:D
461  \__kernel_primitive:NN \maxdepth              \tex_maxdepth:D
462  \__kernel_primitive:NN \meaning               \tex_meaning:D
463  \__kernel_primitive:NN \medmuskip             \tex_medmuskip:D
464  \__kernel_primitive:NN \message               \tex_message:D
465  \__kernel_primitive:NN \mkern                 \tex_mkern:D
466  \__kernel_primitive:NN \month                 \tex_month:D
467  \__kernel_primitive:NN \moveleft              \tex_moveleft:D
```

233

```
468  \__kernel_primitive:NN \moveright                    \tex_moveright:D
469  \__kernel_primitive:NN \mskip                        \tex_mskip:D
470  \__kernel_primitive:NN \multiply                     \tex_multiply:D
471  \__kernel_primitive:NN \muskip                       \tex_muskip:D
472  \__kernel_primitive:NN \muskipdef                    \tex_muskipdef:D
473  \__kernel_primitive:NN \newlinechar                  \tex_newlinechar:D
474  \__kernel_primitive:NN \noalign                      \tex_noalign:D
475  \__kernel_primitive:NN \noboundary                   \tex_noboundary:D
476  \__kernel_primitive:NN \noexpand                     \tex_noexpand:D
477  \__kernel_primitive:NN \noindent                     \tex_noindent:D
478  \__kernel_primitive:NN \nolimits                     \tex_nolimits:D
479  \__kernel_primitive:NN \nonscript                    \tex_nonscript:D
480  \__kernel_primitive:NN \nonstopmode                  \tex_nonstopmode:D
481  \__kernel_primitive:NN \nulldelimiterspace           \tex_nulldelimiterspace:D
482  \__kernel_primitive:NN \nullfont                     \tex_nullfont:D
483  \__kernel_primitive:NN \number                       \tex_number:D
484  \__kernel_primitive:NN \omit                         \tex_omit:D
485  \__kernel_primitive:NN \openin                       \tex_openin:D
486  \__kernel_primitive:NN \openout                      \tex_openout:D
487  \__kernel_primitive:NN \or                           \tex_or:D
488  \__kernel_primitive:NN \outer                        \tex_outer:D
489  \__kernel_primitive:NN \output                       \tex_output:D
490  \__kernel_primitive:NN \outputpenalty                \tex_outputpenalty:D
491  \__kernel_primitive:NN \over                         \tex_over:D
492  \__kernel_primitive:NN \overfullrule                 \tex_overfullrule:D
493  \__kernel_primitive:NN \overline                     \tex_overline:D
494  \__kernel_primitive:NN \overwithdelims               \tex_overwithdelims:D
495  \__kernel_primitive:NN \pagedepth                    \tex_pagedepth:D
496  \__kernel_primitive:NN \pagefilllstretch             \tex_pagefilllstretch:D
497  \__kernel_primitive:NN \pagefillstretch              \tex_pagefillstretch:D
498  \__kernel_primitive:NN \pagefilstretch               \tex_pagefilstretch:D
499  \__kernel_primitive:NN \pagegoal                     \tex_pagegoal:D
500  \__kernel_primitive:NN \pageshrink                   \tex_pageshrink:D
501  \__kernel_primitive:NN \pagestretch                  \tex_pagestretch:D
502  \__kernel_primitive:NN \pagetotal                    \tex_pagetotal:D
503  \__kernel_primitive:NN \par                          \tex_par:D
504  \__kernel_primitive:NN \parfillskip                  \tex_parfillskip:D
505  \__kernel_primitive:NN \parindent                    \tex_parindent:D
506  \__kernel_primitive:NN \parshape                     \tex_parshape:D
507  \__kernel_primitive:NN \parskip                      \tex_parskip:D
508  \__kernel_primitive:NN \patterns                     \tex_patterns:D
509  \__kernel_primitive:NN \pausing                      \tex_pausing:D
510  \__kernel_primitive:NN \penalty                      \tex_penalty:D
511  \__kernel_primitive:NN \postdisplaypenalty           \tex_postdisplaypenalty:D
512  \__kernel_primitive:NN \predisplaypenalty            \tex_predisplaypenalty:D
513  \__kernel_primitive:NN \predisplaysize               \tex_predisplaysize:D
514  \__kernel_primitive:NN \pretolerance                 \tex_pretolerance:D
515  \__kernel_primitive:NN \prevdepth                    \tex_prevdepth:D
516  \__kernel_primitive:NN \prevgraf                     \tex_prevgraf:D
517  \__kernel_primitive:NN \radical                      \tex_radical:D
518  \__kernel_primitive:NN \raise                        \tex_raise:D
519  \__kernel_primitive:NN \read                         \tex_read:D
520  \__kernel_primitive:NN \relax                        \tex_relax:D
521  \__kernel_primitive:NN \relpenalty                   \tex_relpenalty:D
```

```
522  \__kernel_primitive:NN \right                    \tex_right:D
523  \__kernel_primitive:NN \righthyphenmin           \tex_righthyphenmin:D
524  \__kernel_primitive:NN \rightskip                \tex_rightskip:D
525  \__kernel_primitive:NN \romannumeral             \tex_romannumeral:D
526  \__kernel_primitive:NN \scriptfont               \tex_scriptfont:D
527  \__kernel_primitive:NN \scriptscriptfont         \tex_scriptscriptfont:D
528  \__kernel_primitive:NN \scriptscriptstyle        \tex_scriptscriptstyle:D
529  \__kernel_primitive:NN \scriptspace              \tex_scriptspace:D
530  \__kernel_primitive:NN \scriptstyle              \tex_scriptstyle:D
531  \__kernel_primitive:NN \scrollmode               \tex_scrollmode:D
532  \__kernel_primitive:NN \setbox                   \tex_setbox:D
533  \__kernel_primitive:NN \setlanguage              \tex_setlanguage:D
534  \__kernel_primitive:NN \sfcode                   \tex_sfcode:D
535  \__kernel_primitive:NN \shipout                  \tex_shipout:D
536  \__kernel_primitive:NN \show                     \tex_show:D
537  \__kernel_primitive:NN \showbox                  \tex_showbox:D
538  \__kernel_primitive:NN \showboxbreadth           \tex_showboxbreadth:D
539  \__kernel_primitive:NN \showboxdepth             \tex_showboxdepth:D
540  \__kernel_primitive:NN \showlists                \tex_showlists:D
541  \__kernel_primitive:NN \showthe                  \tex_showthe:D
542  \__kernel_primitive:NN \skewchar                 \tex_skewchar:D
543  \__kernel_primitive:NN \skip                     \tex_skip:D
544  \__kernel_primitive:NN \skipdef                  \tex_skipdef:D
545  \__kernel_primitive:NN \spacefactor              \tex_spacefactor:D
546  \__kernel_primitive:NN \spaceskip                \tex_spaceskip:D
547  \__kernel_primitive:NN \span                     \tex_span:D
548  \__kernel_primitive:NN \special                  \tex_special:D
549  \__kernel_primitive:NN \splitbotmark             \tex_splitbotmark:D
550  \__kernel_primitive:NN \splitfirstmark           \tex_splitfirstmark:D
551  \__kernel_primitive:NN \splitmaxdepth            \tex_splitmaxdepth:D
552  \__kernel_primitive:NN \splittopskip             \tex_splittopskip:D
553  \__kernel_primitive:NN \string                   \tex_string:D
554  \__kernel_primitive:NN \tabskip                  \tex_tabskip:D
555  \__kernel_primitive:NN \textfont                 \tex_textfont:D
556  \__kernel_primitive:NN \textstyle                \tex_textstyle:D
557  \__kernel_primitive:NN \the                      \tex_the:D
558  \__kernel_primitive:NN \thickmuskip              \tex_thickmuskip:D
559  \__kernel_primitive:NN \thinmuskip               \tex_thinmuskip:D
560  \__kernel_primitive:NN \time                     \tex_time:D
561  \__kernel_primitive:NN \toks                     \tex_toks:D
562  \__kernel_primitive:NN \toksdef                  \tex_toksdef:D
563  \__kernel_primitive:NN \tolerance                \tex_tolerance:D
564  \__kernel_primitive:NN \topmark                  \tex_topmark:D
565  \__kernel_primitive:NN \topskip                  \tex_topskip:D
566  \__kernel_primitive:NN \tracingcommands          \tex_tracingcommands:D
567  \__kernel_primitive:NN \tracinglostchars         \tex_tracinglostchars:D
568  \__kernel_primitive:NN \tracingmacros            \tex_tracingmacros:D
569  \__kernel_primitive:NN \tracingonline            \tex_tracingonline:D
570  \__kernel_primitive:NN \tracingoutput            \tex_tracingoutput:D
571  \__kernel_primitive:NN \tracingpages             \tex_tracingpages:D
572  \__kernel_primitive:NN \tracingparagraphs        \tex_tracingparagraphs:D
573  \__kernel_primitive:NN \tracingrestores          \tex_tracingrestores:D
574  \__kernel_primitive:NN \tracingstats             \tex_tracingstats:D
575  \__kernel_primitive:NN \uccode                   \tex_uccode:D
```

```
576  \__kernel_primitive:NN \uchyph                          \tex_uchyph:D
577  \__kernel_primitive:NN \underline                       \tex_underline:D
578  \__kernel_primitive:NN \unhbox                          \tex_unhbox:D
579  \__kernel_primitive:NN \unhcopy                         \tex_unhcopy:D
580  \__kernel_primitive:NN \unkern                          \tex_unkern:D
581  \__kernel_primitive:NN \unpenalty                       \tex_unpenalty:D
582  \__kernel_primitive:NN \unskip                          \tex_unskip:D
583  \__kernel_primitive:NN \unvbox                          \tex_unvbox:D
584  \__kernel_primitive:NN \unvcopy                         \tex_unvcopy:D
585  \__kernel_primitive:NN \uppercase                       \tex_uppercase:D
586  \__kernel_primitive:NN \vadjust                         \tex_vadjust:D
587  \__kernel_primitive:NN \valign                          \tex_valign:D
588  \__kernel_primitive:NN \vbadness                        \tex_vbadness:D
589  \__kernel_primitive:NN \vbox                            \tex_vbox:D
590  \__kernel_primitive:NN \vcenter                         \tex_vcenter:D
591  \__kernel_primitive:NN \vfil                            \tex_vfil:D
592  \__kernel_primitive:NN \vfill                           \tex_vfill:D
593  \__kernel_primitive:NN \vfilneg                         \tex_vfilneg:D
594  \__kernel_primitive:NN \vfuzz                           \tex_vfuzz:D
595  \__kernel_primitive:NN \voffset                         \tex_voffset:D
596  \__kernel_primitive:NN \vrule                           \tex_vrule:D
597  \__kernel_primitive:NN \vsize                           \tex_vsize:D
598  \__kernel_primitive:NN \vskip                           \tex_vskip:D
599  \__kernel_primitive:NN \vsplit                          \tex_vsplit:D
600  \__kernel_primitive:NN \vss                             \tex_vss:D
601  \__kernel_primitive:NN \vtop                            \tex_vtop:D
602  \__kernel_primitive:NN \wd                              \tex_wd:D
603  \__kernel_primitive:NN \widowpenalty                    \tex_widowpenalty:D
604  \__kernel_primitive:NN \write                           \tex_write:D
605  \__kernel_primitive:NN \xdef                            \tex_xdef:D
606  \__kernel_primitive:NN \xleaders                        \tex_xleaders:D
607  \__kernel_primitive:NN \xspaceskip                      \tex_xspaceskip:D
608  \__kernel_primitive:NN \year                            \tex_year:D
```

Since LATEX3 requires at least the ε-TEX extensions, we also rename the additional primitives. These are all given the prefix \etex_.

```
609  \__kernel_primitive:NN \beginL                          \etex_beginL:D
610  \__kernel_primitive:NN \beginR                          \etex_beginR:D
611  \__kernel_primitive:NN \botmarks                        \etex_botmarks:D
612  \__kernel_primitive:NN \clubpenalties                   \etex_clubpenalties:D
613  \__kernel_primitive:NN \currentgrouplevel               \etex_currentgrouplevel:D
614  \__kernel_primitive:NN \currentgrouptype                \etex_currentgrouptype:D
615  \__kernel_primitive:NN \currentifbranch                 \etex_currentifbranch:D
616  \__kernel_primitive:NN \currentiflevel                  \etex_currentiflevel:D
617  \__kernel_primitive:NN \currentiftype                   \etex_currentiftype:D
618  \__kernel_primitive:NN \detokenize                      \etex_detokenize:D
619  \__kernel_primitive:NN \dimexpr                         \etex_dimexpr:D
620  \__kernel_primitive:NN \displaywidowpenalties           \etex_displaywidowpenalties:D
621  \__kernel_primitive:NN \endL                            \etex_endL:D
622  \__kernel_primitive:NN \endR                            \etex_endR:D
623  \__kernel_primitive:NN \eTeXrevision                    \etex_eTeXrevision:D
624  \__kernel_primitive:NN \eTeXversion                     \etex_eTeXversion:D
625  \__kernel_primitive:NN \everyeof                        \etex_everyeof:D
626  \__kernel_primitive:NN \firstmarks                      \etex_firstmarks:D
```

236

627	`\__kernel_primitive:NN \fontchardp`	`\etex_fontchardp:D`
628	`\__kernel_primitive:NN \fontcharht`	`\etex_fontcharht:D`
629	`\__kernel_primitive:NN \fontcharic`	`\etex_fontcharic:D`
630	`\__kernel_primitive:NN \fontcharwd`	`\etex_fontcharwd:D`
631	`\__kernel_primitive:NN \glueexpr`	`\etex_glueexpr:D`
632	`\__kernel_primitive:NN \glueshrink`	`\etex_glueshrink:D`
633	`\__kernel_primitive:NN \glueshrinkorder`	`\etex_glueshrinkorder:D`
634	`\__kernel_primitive:NN \gluestretch`	`\etex_gluestretch:D`
635	`\__kernel_primitive:NN \gluestretchorder`	`\etex_gluestretchorder:D`
636	`\__kernel_primitive:NN \gluetomu`	`\etex_gluetomu:D`
637	`\__kernel_primitive:NN \ifcsname`	`\etex_ifcsname:D`
638	`\__kernel_primitive:NN \ifdefined`	`\etex_ifdefined:D`
639	`\__kernel_primitive:NN \iffontchar`	`\etex_iffontchar:D`
640	`\__kernel_primitive:NN \interactionmode`	`\etex_interactionmode:D`
641	`\__kernel_primitive:NN \interlinepenalties`	`\etex_interlinepenalties:D`
642	`\__kernel_primitive:NN \lastlinefit`	`\etex_lastlinefit:D`
643	`\__kernel_primitive:NN \lastnodetype`	`\etex_lastnodetype:D`
644	`\__kernel_primitive:NN \marks`	`\etex_marks:D`
645	`\__kernel_primitive:NN \middle`	`\etex_middle:D`
646	`\__kernel_primitive:NN \muexpr`	`\etex_muexpr:D`
647	`\__kernel_primitive:NN \mutoglue`	`\etex_mutoglue:D`
648	`\__kernel_primitive:NN \numexpr`	`\etex_numexpr:D`
649	`\__kernel_primitive:NN \pagediscards`	`\etex_pagediscards:D`
650	`\__kernel_primitive:NN \parshapedimen`	`\etex_parshapedimen:D`
651	`\__kernel_primitive:NN \parshapeindent`	`\etex_parshapeindent:D`
652	`\__kernel_primitive:NN \parshapelength`	`\etex_parshapelength:D`
653	`\__kernel_primitive:NN \predisplaydirection`	`\etex_predisplaydirection:D`
654	`\__kernel_primitive:NN \protected`	`\etex_protected:D`
655	`\__kernel_primitive:NN \readline`	`\etex_readline:D`
656	`\__kernel_primitive:NN \savinghyphcodes`	`\etex_savinghyphcodes:D`
657	`\__kernel_primitive:NN \savingvdiscards`	`\etex_savingvdiscards:D`
658	`\__kernel_primitive:NN \scantokens`	`\etex_scantokens:D`
659	`\__kernel_primitive:NN \showgroups`	`\etex_showgroups:D`
660	`\__kernel_primitive:NN \showifs`	`\etex_showifs:D`
661	`\__kernel_primitive:NN \showtokens`	`\etex_showtokens:D`
662	`\__kernel_primitive:NN \splitbotmarks`	`\etex_splitbotmarks:D`
663	`\__kernel_primitive:NN \splitdiscards`	`\etex_splitdiscards:D`
664	`\__kernel_primitive:NN \splitfirstmarks`	`\etex_splitfirstmarks:D`
665	`\__kernel_primitive:NN \TeXXeTstate`	`\etex_TeXXeTstate:D`
666	`\__kernel_primitive:NN \topmarks`	`\etex_topmarks:D`
667	`\__kernel_primitive:NN \tracingassigns`	`\etex_tracingassigns:D`
668	`\__kernel_primitive:NN \tracinggroups`	`\etex_tracinggroups:D`
669	`\__kernel_primitive:NN \tracingifs`	`\etex_tracingifs:D`
670	`\__kernel_primitive:NN \tracingnesting`	`\etex_tracingnesting:D`
671	`\__kernel_primitive:NN \tracingscantokens`	`\etex_tracingscantokens:D`
672	`\__kernel_primitive:NN \unexpanded`	`\etex_unexpanded:D`
673	`\__kernel_primitive:NN \unless`	`\etex_unless:D`
674	`\__kernel_primitive:NN \widowpenalties`	`\etex_widowpenalties:D`

The newer primitives are more complex: there are an awful lot of them, and we don't use them all at the moment. So the following is selective, based on those also available in LuaTeX or used in expl3. In the case of the pdfTeX primitives, we retain `pdf` at the start of the names *only* for directly PDF-related primitives, as there are a lot of pdfTeX primitives that start `\pdf...` but are not related to PDF output. These ones related to

PDF output or only work in PDF mode.

```
675  \__kernel_primitive:NN \pdfannot                    \pdftex_pdfannot:D
676  \__kernel_primitive:NN \pdfcatalog                  \pdftex_pdfcatalog:D
677  \__kernel_primitive:NN \pdfcompresslevel            \pdftex_pdfcompresslevel:D
678  \__kernel_primitive:NN \pdfcolorstack               \pdftex_pdfcolorstack:D
679  \__kernel_primitive:NN \pdfcolorstackinit           \pdftex_pdfcolorstackinit:D
680  \__kernel_primitive:NN \pdfcreationdate             \pdftex_pdfcreationdate:D
681  \__kernel_primitive:NN \pdfdecimaldigits            \pdftex_pdfdecimaldigits:D
682  \__kernel_primitive:NN \pdfdest                     \pdftex_pdfdest:D
683  \__kernel_primitive:NN \pdfdestmargin               \pdftex_pdfdestmargin:D
684  \__kernel_primitive:NN \pdfendlink                  \pdftex_pdfendlink:D
685  \__kernel_primitive:NN \pdfendthread                \pdftex_pdfendthread:D
686  \__kernel_primitive:NN \pdffontattr                 \pdftex_pdffontattr:D
687  \__kernel_primitive:NN \pdffontname                 \pdftex_pdffontname:D
688  \__kernel_primitive:NN \pdffontobjnum               \pdftex_pdffontobjnum:D
689  \__kernel_primitive:NN \pdfgamma                    \pdftex_pdfgamma:D
690  \__kernel_primitive:NN \pdfimageapplygamma          \pdftex_pdfimageapplygamma:D
691  \__kernel_primitive:NN \pdfimagegamma               \pdftex_pdfimagegamma:D
692  \__kernel_primitive:NN \pdfgentounicode             \pdftex_pdfgentounicode:D
693  \__kernel_primitive:NN \pdfglyphtounicode           \pdftex_pdfglyphtounicode:D
694  \__kernel_primitive:NN \pdfhorigin                  \pdftex_pdfhorigin:D
695  \__kernel_primitive:NN \pdfimagehicolor             \pdftex_pdfimagehicolor:D
696  \__kernel_primitive:NN \pdfimageresolution          \pdftex_pdfimageresolution:D
697  \__kernel_primitive:NN \pdfincludechars             \pdftex_pdfincludechars:D
698  \__kernel_primitive:NN \pdfinclusioncopyfonts       \pdftex_pdfinclusioncopyfonts:D
699  \__kernel_primitive:NN \pdfinclusionerrorlevel      \pdftex_pdfinclusionerrorlevel:D
700  \__kernel_primitive:NN \pdfinfo                     \pdftex_pdfinfo:D
701  \__kernel_primitive:NN \pdflastannot                \pdftex_pdflastannot:D
702  \__kernel_primitive:NN \pdflastlink                 \pdftex_pdflastlink:D
703  \__kernel_primitive:NN \pdflastobj                  \pdftex_pdflastobj:D
704  \__kernel_primitive:NN \pdflastxform                \pdftex_pdflastxform:D
705  \__kernel_primitive:NN \pdflastximage               \pdftex_pdflastximage:D
706  \__kernel_primitive:NN \pdflastximagecolordepth     \pdftex_pdflastximagecolordepth:D
707  \__kernel_primitive:NN \pdflastximagepages          \pdftex_pdflastximagepages:D
708  \__kernel_primitive:NN \pdflinkmargin               \pdftex_pdflinkmargin:D
709  \__kernel_primitive:NN \pdfliteral                  \pdftex_pdfliteral:D
710  \__kernel_primitive:NN \pdfminorversion             \pdftex_pdfminorversion:D
711  \__kernel_primitive:NN \pdfnames                    \pdftex_pdfnames:D
712  \__kernel_primitive:NN \pdfobj                      \pdftex_pdfobj:D
713  \__kernel_primitive:NN \pdfobjcompresslevel         \pdftex_pdfobjcompresslevel:D
714  \__kernel_primitive:NN \pdfoutline                  \pdftex_pdfoutline:D
715  \__kernel_primitive:NN \pdfoutput                   \pdftex_pdfoutput:D
716  \__kernel_primitive:NN \pdfpageattr                 \pdftex_pdfpageattr:D
717  \__kernel_primitive:NN \pdfpagebox                  \pdftex_pdfpagebox:D
718  \__kernel_primitive:NN \pdfpageref                  \pdftex_pdfpageref:D
719  \__kernel_primitive:NN \pdfpageresources            \pdftex_pdfpageresources:D
720  \__kernel_primitive:NN \pdfpagesattr                \pdftex_pdfpagesattr:D
721  \__kernel_primitive:NN \pdfrefobj                   \pdftex_pdfrefobj:D
722  \__kernel_primitive:NN \pdfrefxform                 \pdftex_pdfrefxform:D
723  \__kernel_primitive:NN \pdfrefximage                \pdftex_pdfrefximage:D
724  \__kernel_primitive:NN \pdfrestore                  \pdftex_pdfrestore:D
725  \__kernel_primitive:NN \pdfretval                   \pdftex_pdfretval:D
726  \__kernel_primitive:NN \pdfsave                     \pdftex_pdfsave:D
727  \__kernel_primitive:NN \pdfsetmatrix                \pdftex_pdfsetmatrix:D
```

```
728    \__kernel_primitive:NN \pdfstartlink              \pdftex_pdfstartlink:D
729    \__kernel_primitive:NN \pdfstartthread            \pdftex_pdfstartthread:D
730    \__kernel_primitive:NN \pdfsuppressptexinfo       \pdftex_pdfsuppressptexinfo:D
731    \__kernel_primitive:NN \pdfthread                 \pdftex_pdfthread:D
732    \__kernel_primitive:NN \pdfthreadmargin           \pdftex_pdfthreadmargin:D
733    \__kernel_primitive:NN \pdftrailer                \pdftex_pdftrailer:D
734    \__kernel_primitive:NN \pdfuniqueresname          \pdftex_pdfuniqueresname:D
735    \__kernel_primitive:NN \pdfvorigin                \pdftex_pdfvorigin:D
736    \__kernel_primitive:NN \pdfxform                  \pdftex_pdfxform:D
737    \__kernel_primitive:NN \pdfxformattr              \pdftex_pdfxformattr:D
738    \__kernel_primitive:NN \pdfxformname              \pdftex_pdfxformname:D
739    \__kernel_primitive:NN \pdfxformresources         \pdftex_pdfxformresources:D
740    \__kernel_primitive:NN \pdfximage                 \pdftex_pdfximage:D
741    \__kernel_primitive:NN \pdfximagebbox             \pdftex_pdfximagebbox:D
```

While these are not.

```
742    \__kernel_primitive:NN \ifpdfabsdim               \pdftex_ifabsdim:D
743    \__kernel_primitive:NN \ifpdfabsnum               \pdftex_ifabsnum:D
744    \__kernel_primitive:NN \ifpdfprimitive            \pdftex_ifprimitive:D
745    \__kernel_primitive:NN \pdfadjustspacing          \pdftex_adjustspacing:D
746    \__kernel_primitive:NN \pdfcopyfont               \pdftex_copyfont:D
747    \__kernel_primitive:NN \pdfdraftmode              \pdftex_draftmode:D
748    \__kernel_primitive:NN \pdfeachlinedepth          \pdftex_eachlinedepth:D
749    \__kernel_primitive:NN \pdfeachlineheight         \pdftex_eachlineheight:D
750    \__kernel_primitive:NN \pdffirstlineheight        \pdftex_firstlineheight:D
751    \__kernel_primitive:NN \pdffontexpand             \pdftex_fontexpand:D
752    \__kernel_primitive:NN \pdffontsize               \pdftex_fontsize:D
753    \__kernel_primitive:NN \pdfignoreddimen           \pdftex_ignoreddimen:D
754    \__kernel_primitive:NN \pdfinsertht               \pdftex_insertht:D
755    \__kernel_primitive:NN \pdflastlinedepth          \pdftex_lastlinedepth:D
756    \__kernel_primitive:NN \pdflastxpos               \pdftex_lastxpos:D
757    \__kernel_primitive:NN \pdflastypos               \pdftex_lastypos:D
758    \__kernel_primitive:NN \pdfmapfile                \pdftex_mapfile:D
759    \__kernel_primitive:NN \pdfmapline                \pdftex_mapline:D
760    \__kernel_primitive:NN \pdfnoligatures            \pdftex_noligatures:D
761    \__kernel_primitive:NN \pdfnormaldeviate          \pdftex_normaldeviate:D
762    \__kernel_primitive:NN \pdfpageheight             \pdftex_pageheight:D
763    \__kernel_primitive:NN \pdfpagewidth              \pdftex_pagewidth:D
764    \__kernel_primitive:NN \pdfpkmode                 \pdftex_pkmode:D
765    \__kernel_primitive:NN \pdfpkresolution           \pdftex_pkresolution:D
766    \__kernel_primitive:NN \pdfprimitive              \pdftex_primitive:D
767    \__kernel_primitive:NN \pdfprotrudechars          \pdftex_protrudechars:D
768    \__kernel_primitive:NN \pdfpxdimen                \pdftex_pxdimen:D
769    \__kernel_primitive:NN \pdfrandomseed             \pdftex_randomseed:D
770    \__kernel_primitive:NN \pdfsavepos                \pdftex_savepos:D
771    \__kernel_primitive:NN \pdfstrcmp                 \pdftex_strcmp:D
772    \__kernel_primitive:NN \pdfsetrandomseed          \pdftex_setrandomseed:D
773    \__kernel_primitive:NN \pdfshellescape            \pdftex_shellescape:D
774    \__kernel_primitive:NN \pdftracingfonts           \pdftex_tracingfonts:D
775    \__kernel_primitive:NN \pdfuniformdeviate         \pdftex_uniformdeviate:D
```

The version primitives are not related to PDF mode but are related to pdfTeX so retain the full prefix.

```
776    \__kernel_primitive:NN \pdftexbanner              \pdftex_pdftexbanner:D
777    \__kernel_primitive:NN \pdftexrevision            \pdftex_pdftexrevision:D
```

```
778  \__kernel_primitive:NN \pdftexversion              \pdftex_pdftexversion:D
```

These ones appear in pdfTeX but don't have `pdf` in the name at all. (`\synctex` is odd as it's really not from pdfTeX but from SyncTeX!)

```
779  \__kernel_primitive:NN \efcode                     \pdftex_efcode:D
780  \__kernel_primitive:NN \ifincsname                 \pdftex_ifincsname:D
781  \__kernel_primitive:NN \leftmarginkern             \pdftex_leftmarginkern:D
782  \__kernel_primitive:NN \letterspacefont            \pdftex_letterspacefont:D
783  \__kernel_primitive:NN \lpcode                     \pdftex_lpcode:D
784  \__kernel_primitive:NN \quitvmode                  \pdftex_quitvmode:D
785  \__kernel_primitive:NN \rightmarginkern            \pdftex_rightmarginkern:D
786  \__kernel_primitive:NN \rpcode                     \pdftex_rpcode:D
787  \__kernel_primitive:NN \synctex                    \pdftex_synctex:D
788  \__kernel_primitive:NN \tagcode                    \pdftex_tagcode:D
```

Post pdfTeX primitive availability gets more complex. Both XƎTeX 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.

```
789  ⟨/initex | names | package⟩
790  ⟨*initex | package⟩
791    \tex_long:D \tex_def:D \use_ii:nn #1#2 {#2}
792    \tex_long:D \tex_def:D \use_none:n #1 { }
793    \tex_long:D \tex_def:D \__kernel_primitive:NN #1#2
794      {
795        \etex_ifdefined:D #1
796          \tex_expandafter:D \use_ii:nn
797        \tex_fi:
798          \use_none:n { \tex_global:D \tex_let:D #2 #1 }
799  ⟨*initex⟩
800        \tex_global:D \tex_let:D #1 \tex_undefined:D
801  ⟨/initex⟩
802      }
803  ⟨/initex | package⟩
804  ⟨*initex | names | package⟩
```

XƎTeX-specific primitives. Note that XƎTeX's `\strcmp` is handled earlier and is "rolled up" into `\pdfstrcmp`. With the exception of the version primitives these don't carry `XeTeX` through into the "base" name. A few cross-compatibility names which lack the `pdf` of the original are handled later.

```
805  \__kernel_primitive:NN \suppressfontnotfounderror   \xetex_suppressfontnotfounderror:D
806  \__kernel_primitive:NN \XeTeXcharclass              \xetex_charclass:D
807  \__kernel_primitive:NN \XeTeXcharglyph              \xetex_charglyph:D
808  \__kernel_primitive:NN \XeTeXcountfeatures          \xetex_countfeatures:D
809  \__kernel_primitive:NN \XeTeXcountglyphs            \xetex_countglyphs:D
810  \__kernel_primitive:NN \XeTeXcountselectors         \xetex_countselectors:D
811  \__kernel_primitive:NN \XeTeXcountvariations        \xetex_countvariations:D
812  \__kernel_primitive:NN \XeTeXdefaultencoding        \xetex_defaultencoding:D
813  \__kernel_primitive:NN \XeTeXdashbreakstate         \xetex_dashbreakstate:D
814  \__kernel_primitive:NN \XeTeXfeaturecode            \xetex_featurecode:D
815  \__kernel_primitive:NN \XeTeXfeaturename            \xetex_featurename:D
816  \__kernel_primitive:NN \XeTeXfindfeaturebyname      \xetex_findfeaturebyname:D
817  \__kernel_primitive:NN \XeTeXfindselectorbyname     \xetex_findselectorbyname:D
818  \__kernel_primitive:NN \XeTeXfindvariationbyname    \xetex_findvariationbyname:D
819  \__kernel_primitive:NN \XeTeXfirstfontchar          \xetex_firstfontchar:D
```

```
820    \__kernel_primitive:NN \XeTeXfonttype              \xetex_fonttype:D
821    \__kernel_primitive:NN \XeTeXgenerateactualtext    \xetex_generateactualtext:D
822    \__kernel_primitive:NN \XeTeXglyph                 \xetex_glyph:D
823    \__kernel_primitive:NN \XeTeXglyphbounds           \xetex_glyphbounds:D
824    \__kernel_primitive:NN \XeTeXglyphindex            \xetex_glyphindex:D
825    \__kernel_primitive:NN \XeTeXglyphname             \xetex_glyphname:D
826    \__kernel_primitive:NN \XeTeXinputencoding         \xetex_inputencoding:D
827    \__kernel_primitive:NN \XeTeXinputnormalization    \xetex_inputnormalization:D
828    \__kernel_primitive:NN \XeTeXinterchartokenstate   \xetex_interchartokenstate:D
829    \__kernel_primitive:NN \XeTeXinterchartoks         \xetex_interchartoks:D
830    \__kernel_primitive:NN \XeTeXisdefaultselector     \xetex_isdefaultselector:D
831    \__kernel_primitive:NN \XeTeXisexclusivefeature    \xetex_isexclusivefeature:D
832    \__kernel_primitive:NN \XeTeXlastfontchar          \xetex_lastfontchar:D
833    \__kernel_primitive:NN \XeTeXlinebreakskip         \xetex_linebreakskip:D
834    \__kernel_primitive:NN \XeTeXlinebreaklocale       \xetex_linebreaklocale:D
835    \__kernel_primitive:NN \XeTeXlinebreakpenalty      \xetex_linebreakpenalty:D
836    \__kernel_primitive:NN \XeTeXOTcountfeatures       \xetex_OTcountfeatures:D
837    \__kernel_primitive:NN \XeTeXOTcountlanguages      \xetex_OTcountlanguages:D
838    \__kernel_primitive:NN \XeTeXOTcountscripts        \xetex_OTcountscripts:D
839    \__kernel_primitive:NN \XeTeXOTfeaturetag          \xetex_OTfeaturetag:D
840    \__kernel_primitive:NN \XeTeXOTlanguagetag         \xetex_OTlanguagetag:D
841    \__kernel_primitive:NN \XeTeXOTscripttag           \xetex_OTscripttag:D
842    \__kernel_primitive:NN \XeTeXpdffile               \xetex_pdffile:D
843    \__kernel_primitive:NN \XeTeXpdfpagecount          \xetex_pdfpagecount:D
844    \__kernel_primitive:NN \XeTeXpicfile               \xetex_picfile:D
845    \__kernel_primitive:NN \XeTeXselectorname          \xetex_selectorname:D
846    \__kernel_primitive:NN \XeTeXtracingfonts          \xetex_tracingfonts:D
847    \__kernel_primitive:NN \XeTeXupwardsmode           \xetex_upwardsmode:D
848    \__kernel_primitive:NN \XeTeXuseglyphmetrics       \xetex_useglyphmetrics:D
849    \__kernel_primitive:NN \XeTeXvariation             \xetex_variation:D
850    \__kernel_primitive:NN \XeTeXvariationdefault      \xetex_variationdefault:D
851    \__kernel_primitive:NN \XeTeXvariationmax          \xetex_variationmax:D
852    \__kernel_primitive:NN \XeTeXvariationmin          \xetex_variationmin:D
853    \__kernel_primitive:NN \XeTeXvariationname         \xetex_variationname:D
```

The version primitives retain `XeTeX`.

```
854    \__kernel_primitive:NN \XeTeXrevision              \xetex_XeTeXrevision:D
855    \__kernel_primitive:NN \XeTeXversion               \xetex_XeTeXversion:D
```

Primitives from pdfTeX that X$_{\overline{E}}$TeX renames: also helps with LuaTeX.

```
856    \__kernel_primitive:NN \ifprimitive                \pdftex_ifprimitive:D
857    \__kernel_primitive:NN \primitive                  \pdftex_primitive:D
858    \__kernel_primitive:NN \shellescape                \pdftex_shellescape:D
```

Primitives from LuaTeX, some of which have been ported back to X$_{\overline{E}}$TeX. Notice that `\expanded` was intended for pdfTeX 1.50 but as that was not released we call this a LuaTeX primitive.

```
859    \__kernel_primitive:NN \alignmark                  \luatex_alignmark:D
860    \__kernel_primitive:NN \aligntab                   \luatex_aligntab:D
861    \__kernel_primitive:NN \attribute                  \luatex_attribute:D
862    \__kernel_primitive:NN \attributedef               \luatex_attributedef:D
863    \__kernel_primitive:NN \begincsname                \luatex_begincsname:D
864    \__kernel_primitive:NN \catcodetable               \luatex_catcodetable:D
865    \__kernel_primitive:NN \clearmarks                 \luatex_clearmarks:D
866    \__kernel_primitive:NN \crampeddisplaystyle        \luatex_crampeddisplaystyle:D
```

```
867  \__kernel_primitive:NN \crampedscriptscriptstyle        \luatex_crampedscriptscriptstyle:D
868  \__kernel_primitive:NN \crampedscriptstyle              \luatex_crampedscriptstyle:D
869  \__kernel_primitive:NN \crampedtextstyle                \luatex_crampedtextstyle:D
870  \__kernel_primitive:NN \directlua                       \luatex_directlua:D
871  \__kernel_primitive:NN \dviextension                    \luatex_dviextension:D
872  \__kernel_primitive:NN \dvifeedback                     \luatex_dvifeedback:D
873  \__kernel_primitive:NN \dvivariable                     \luatex_dvivariable:D
874  \__kernel_primitive:NN \etoksapp                        \luatex_etoksapp:D
875  \__kernel_primitive:NN \etokspre                        \luatex_etokspre:D
876  \__kernel_primitive:NN \expanded                        \luatex_expanded:D
877  \__kernel_primitive:NN \firstvalidlanguage              \luatex_firstvalidlanguage:D
878  \__kernel_primitive:NN \fontid                          \luatex_fontid:D
879  \__kernel_primitive:NN \formatname                      \luatex_formatname:D
880  \__kernel_primitive:NN \hjcode                          \luatex_hjcode:D
881  \__kernel_primitive:NN \hpack                           \luatex_hpack:D
882  \__kernel_primitive:NN \hyphenationbounds               \luatex_hyphenationbounds:D
883  \__kernel_primitive:NN \hyphenationmin                  \luatex_hyphenationmin:D
884  \__kernel_primitive:NN \gleaders                        \luatex_gleaders:D
885  \__kernel_primitive:NN \initcatcodetable                \luatex_initcatcodetable:D
886  \__kernel_primitive:NN \lastnamedcs                     \luatex_lastnamedcs:D
887  \__kernel_primitive:NN \latelua                         \luatex_latelua:D
888  \__kernel_primitive:NN \letcharcode                     \luatex_letcharcode:D
889  \__kernel_primitive:NN \luaescapestring                 \luatex_luaescapestring:D
890  \__kernel_primitive:NN \luafunction                     \luatex_luafunction:D
891  \__kernel_primitive:NN \luatexbanner                    \luatex_luatexbanner:D
892  \__kernel_primitive:NN \luatexdatestamp                 \luatex_luatexdatestamp:D
893  \__kernel_primitive:NN \luatexrevision                  \luatex_luatexrevision:D
894  \__kernel_primitive:NN \luatexversion                   \luatex_luatexversion:D
895  \__kernel_primitive:NN \mathdisplayskipmode             \luatex_mathdisplayskipmode:D
896  \__kernel_primitive:NN \matheqnogapstep                 \luatex_matheqnogapstep:D
897  \__kernel_primitive:NN \mathoption                      \luatex_mathoption:D
898  \__kernel_primitive:NN \mathnolimitsmode                \luatex_mathnolimitsmode:D
899  \__kernel_primitive:NN \mathrulesfam                    \luatex_mathrulesfam:D
900  \__kernel_primitive:NN \mathscriptsmode                 \luatex_mathscriptsmode:D
901  \__kernel_primitive:NN \mathstyle                       \luatex_mathstyle:D
902  \__kernel_primitive:NN \mathsurroundskip                \luatex_mathsurroundskip:D
903  \__kernel_primitive:NN \nohrule                         \luatex_nohrule:D
904  \__kernel_primitive:NN \nokerns                         \luatex_nokerns:D
905  \__kernel_primitive:NN \noligs                          \luatex_noligs:D
906  \__kernel_primitive:NN \nospaces                        \luatex_nospaces:D
907  \__kernel_primitive:NN \novrule                         \luatex_novrule:D
908  \__kernel_primitive:NN \outputbox                       \luatex_outputbox:D
909  \__kernel_primitive:NN \pageleftoffset                  \luatex_pageleftoffset:D
910  \__kernel_primitive:NN \pagetopoffset                   \luatex_pagetopoffset:D
911  \__kernel_primitive:NN \pdfextension                    \luatex_pdfextension:D
912  \__kernel_primitive:NN \pdffeedback                     \luatex_pdffeedback:D
913  \__kernel_primitive:NN \pdfvariable                     \luatex_pdfvariable:D
914  \__kernel_primitive:NN \postexhyphenchar                \luatex_postexhyphenchar:D
915  \__kernel_primitive:NN \posthyphenchar                  \luatex_posthyphenchar:D
916  \__kernel_primitive:NN \preexhyphenchar                 \luatex_preexhyphenchar:D
917  \__kernel_primitive:NN \prehyphenchar                   \luatex_prehyphenchar:D
918  \__kernel_primitive:NN \savecatcodetable                \luatex_savecatcodetable:D
919  \__kernel_primitive:NN \scantextokens                   \luatex_scantextokens:D
920  \__kernel_primitive:NN \setfontid                       \luatex_setfontid:D
```

```
921  \__kernel_primitive:NN \shapemode              \luatex_shapemode:D
922  \__kernel_primitive:NN \suppressifcsnameerror  \luatex_suppressifcsnameerror:D
923  \__kernel_primitive:NN \suppresslongerror      \luatex_suppresslongerror:D
924  \__kernel_primitive:NN \suppressmathparerror   \luatex_suppressmathparerror:D
925  \__kernel_primitive:NN \suppressoutererror     \luatex_suppressoutererror:D
926  \__kernel_primitive:NN \toksapp               \luatex_toksapp:D
927  \__kernel_primitive:NN \tokspre               \luatex_tokspre:D
928  \__kernel_primitive:NN \tpack                 \luatex_tpack:D
929  \__kernel_primitive:NN \vpack                 \luatex_vpack:D
```

Slightly more awkward are the directional primitives in LuaTeX. These come from Omega/Aleph, but we do not support those engines and so it seems most sensible to treat them as LuaTeX primitives for prefix purposes. One here is "new" but fits into the general set.

```
930  \__kernel_primitive:NN \bodydir               \luatex_bodydir:D
931  \__kernel_primitive:NN \boxdir                \luatex_boxdir:D
932  \__kernel_primitive:NN \leftghost             \luatex_leftghost:D
933  \__kernel_primitive:NN \localbrokenpenalty    \luatex_localbrokenpenalty:D
934  \__kernel_primitive:NN \localinterlinepenalty \luatex_localinterlinepenalty:D
935  \__kernel_primitive:NN \localleftbox          \luatex_localleftbox:D
936  \__kernel_primitive:NN \localrightbox         \luatex_localrightbox:D
937  \__kernel_primitive:NN \mathdir               \luatex_mathdir:D
938  \__kernel_primitive:NN \linedir               \luatex_linedir:D
939  \__kernel_primitive:NN \pagebottomoffset      \luatex_pagebottomoffset:D
940  \__kernel_primitive:NN \pagedir               \luatex_pagedir:D
941  \__kernel_primitive:NN \pagerightoffset       \luatex_pagerightoffset:D
942  \__kernel_primitive:NN \pardir                \luatex_pardir:D
943  \__kernel_primitive:NN \rightghost            \luatex_rightghost:D
944  \__kernel_primitive:NN \textdir               \luatex_textdir:D
```

Primitives from pdfTeX that LuaTeX renames.

```
945  \__kernel_primitive:NN \adjustspacing         \pdftex_adjustspacing:D
946  \__kernel_primitive:NN \copyfont              \pdftex_copyfont:D
947  \__kernel_primitive:NN \draftmode             \pdftex_draftmode:D
948  \__kernel_primitive:NN \expandglyphsinfont    \pdftex_fontexpand:D
949  \__kernel_primitive:NN \ifabsdim              \pdftex_ifabsdim:D
950  \__kernel_primitive:NN \ifabsnum              \pdftex_ifabsnum:D
951  \__kernel_primitive:NN \ignoreligaturesinfont \pdftex_ignoreligaturesinfont:D
952  \__kernel_primitive:NN \insertht             \pdftex_insertht:D
953  \__kernel_primitive:NN \lastsavedboxresourceindex \pdftex_pdflastxform:D
954  \__kernel_primitive:NN \lastsavedimageresourceindex \pdftex_pdflastximage:D
955  \__kernel_primitive:NN \lastsavedimageresourcepages \pdftex_pdflastximagepages:D
956  \__kernel_primitive:NN \lastxpos             \pdftex_lastxpos:D
957  \__kernel_primitive:NN \lastypos             \pdftex_lastypos:D
958  \__kernel_primitive:NN \normaldeviate        \pdftex_normaldeviate:D
959  \__kernel_primitive:NN \outputmode           \pdftex_pdfoutput:D
960  \__kernel_primitive:NN \pageheight           \pdftex_pageheight:D
961  \__kernel_primitive:NN \pagewidth            \pdftex_pagewith:D
962  \__kernel_primitive:NN \protrudechars        \pdftex_protrudechars:D
963  \__kernel_primitive:NN \pxdimen              \pdftex_pxdimen:D
964  \__kernel_primitive:NN \randomseed           \pdftex_randomseed:D
965  \__kernel_primitive:NN \useboxresource       \pdftex_pdfrefxform:D
966  \__kernel_primitive:NN \useimageresource     \pdftex_pdfrefximage:D
967  \__kernel_primitive:NN \savepos              \pdftex_savepos:D
968  \__kernel_primitive:NN \saveboxresource      \pdftex_pdfxform:D
```

The set of Unicode math primitives were introduced by XƎTEX 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....` To keep things somewhat clear we therefore prefix all of these as `\utex...` (introduced by a Unicode TEX engine) and drop `\U(math)` from the names. Where there is a related TEX90 primitive or where it really seems required we keep the `math` part of the name.

244

```
1016  \__kernel_primitive:NN \Umathinnerclosespacing        \utex_innerclosespacing:D
1017  \__kernel_primitive:NN \Umathinnerinnerspacing        \utex_innerinnerspacing:D
1018  \__kernel_primitive:NN \Umathinneropenspacing         \utex_inneropenspacing:D
1019  \__kernel_primitive:NN \Umathinneropspacing           \utex_inneropspacing:D
1020  \__kernel_primitive:NN \Umathinnerordspacing          \utex_innerordspacing:D
1021  \__kernel_primitive:NN \Umathinnerpunctspacing        \utex_innerpunctspacing:D
1022  \__kernel_primitive:NN \Umathinnerrelspacing          \utex_innerrelspacing:D
1023  \__kernel_primitive:NN \Umathlimitabovebgap           \utex_limitabovebgap:D
1024  \__kernel_primitive:NN \Umathlimitabovekern           \utex_limitabovekern:D
1025  \__kernel_primitive:NN \Umathlimitabovevgap           \utex_limitabovevgap:D
1026  \__kernel_primitive:NN \Umathlimitbelowbgap           \utex_limitbelowbgap:D
1027  \__kernel_primitive:NN \Umathlimitbelowkern           \utex_limitbelowkern:D
1028  \__kernel_primitive:NN \Umathlimitbelowvgap           \utex_limitbelowvgap:D
1029  \__kernel_primitive:NN \Umathnolimitsubfactor         \utex_nolimitsubfactor:D
1030  \__kernel_primitive:NN \Umathnolimitsupfactor         \utex_nolimitsupfactor:D
1031  \__kernel_primitive:NN \Umathopbinspacing             \utex_opbinspacing:D
1032  \__kernel_primitive:NN \Umathopclosespacing           \utex_opclosespacing:D
1033  \__kernel_primitive:NN \Umathopenbinspacing           \utex_openbinspacing:D
1034  \__kernel_primitive:NN \Umathopenclosespacing         \utex_openclosespacing:D
1035  \__kernel_primitive:NN \Umathopeninnerspacing         \utex_openinnerspacing:D
1036  \__kernel_primitive:NN \Umathopenopenspacing          \utex_openopenspacing:D
1037  \__kernel_primitive:NN \Umathopenopspacing            \utex_openopspacing:D
1038  \__kernel_primitive:NN \Umathopenordspacing           \utex_openordspacing:D
1039  \__kernel_primitive:NN \Umathopenpunctspacing         \utex_openpunctspacing:D
1040  \__kernel_primitive:NN \Umathopenrelspacing           \utex_openrelspacing:D
1041  \__kernel_primitive:NN \Umathoperatorsize            \utex_operatorsize:D
1042  \__kernel_primitive:NN \Umathopinnerspacing           \utex_opinnerspacing:D
1043  \__kernel_primitive:NN \Umathopopenspacing            \utex_opopenspacing:D
1044  \__kernel_primitive:NN \Umathopopspacing              \utex_opopspacing:D
1045  \__kernel_primitive:NN \Umathopordspacing             \utex_opordspacing:D
1046  \__kernel_primitive:NN \Umathoppunctspacing           \utex_oppunctspacing:D
1047  \__kernel_primitive:NN \Umathoprelspacing             \utex_oprelspacing:D
1048  \__kernel_primitive:NN \Umathordbinspacing            \utex_ordbinspacing:D
1049  \__kernel_primitive:NN \Umathordclosespacing          \utex_ordclosespacing:D
1050  \__kernel_primitive:NN \Umathordinnerspacing          \utex_ordinnerspacing:D
1051  \__kernel_primitive:NN \Umathordopenspacing           \utex_ordopenspacing:D
1052  \__kernel_primitive:NN \Umathordopspacing             \utex_ordopspacing:D
1053  \__kernel_primitive:NN \Umathordordspacing            \utex_ordordspacing:D
1054  \__kernel_primitive:NN \Umathordpunctspacing          \utex_ordpunctspacing:D
1055  \__kernel_primitive:NN \Umathordrelspacing            \utex_ordrelspacing:D
1056  \__kernel_primitive:NN \Umathoverbarkern             \utex_overbarkern:D
1057  \__kernel_primitive:NN \Umathoverbarrule             \utex_overbarrule:D
1058  \__kernel_primitive:NN \Umathoverbarvgap             \utex_overbarvgap:D
1059  \__kernel_primitive:NN \Umathoverdelimiterbgap        \utex_overdelimiterbgap:D
1060  \__kernel_primitive:NN \Umathoverdelimitervgap        \utex_overdelimitervgap:D
1061  \__kernel_primitive:NN \Umathpunctbinspacing          \utex_punctbinspacing:D
1062  \__kernel_primitive:NN \Umathpunctclosespacing        \utex_punctclosespacing:D
1063  \__kernel_primitive:NN \Umathpunctinnerspacing        \utex_punctinnerspacing:D
1064  \__kernel_primitive:NN \Umathpunctopenspacing         \utex_punctopenspacing:D
1065  \__kernel_primitive:NN \Umathpunctopspacing           \utex_punctopspacing:D
1066  \__kernel_primitive:NN \Umathpunctordspacing          \utex_punctordspacing:D
1067  \__kernel_primitive:NN \Umathpunctpunctspacing        \utex_punctpunctspacing:D
1068  \__kernel_primitive:NN \Umathpunctrelspacing          \utex_punctrelspacing:D
1069  \__kernel_primitive:NN \Umathquad                     \utex_quad:D
```

```
1070   \__kernel_primitive:NN \Umathradicaldegreeafter        \utex_radicaldegreeafter:D
1071   \__kernel_primitive:NN \Umathradicaldegreebefore       \utex_radicaldegreebefore:D
1072   \__kernel_primitive:NN \Umathradicaldegreeraise        \utex_radicaldegreeraise:D
1073   \__kernel_primitive:NN \Umathradicalkern               \utex_radicalkern:D
1074   \__kernel_primitive:NN \Umathradicalrule               \utex_radicalrule:D
1075   \__kernel_primitive:NN \Umathradicalvgap               \utex_radicalvgap:D
1076   \__kernel_primitive:NN \Umathrelbinspacing             \utex_relbinspacing:D
1077   \__kernel_primitive:NN \Umathrelclosespacing           \utex_relclosespacing:D
1078   \__kernel_primitive:NN \Umathrelinnerspacing           \utex_relinnerspacing:D
1079   \__kernel_primitive:NN \Umathrelopenspacing            \utex_relopenspacing:D
1080   \__kernel_primitive:NN \Umathrelopspacing              \utex_relopspacing:D
1081   \__kernel_primitive:NN \Umathrelordspacing             \utex_relordspacing:D
1082   \__kernel_primitive:NN \Umathrelpunctspacing           \utex_relpunctspacing:D
1083   \__kernel_primitive:NN \Umathrelrelspacing             \utex_relrelspacing:D
1084   \__kernel_primitive:NN \Umathskewedfractionhgap        \utex_skewedfractionhgap:D
1085   \__kernel_primitive:NN \Umathskewedfractionvgap        \utex_skewedfractionvgap:D
1086   \__kernel_primitive:NN \Umathspaceafterscript          \utex_spaceafterscript:D
1087   \__kernel_primitive:NN \Umathstackdenomdown            \utex_stackdenomdown:D
1088   \__kernel_primitive:NN \Umathstacknumup                \utex_stacknumup:D
1089   \__kernel_primitive:NN \Umathstackvgap                 \utex_stackvgap:D
1090   \__kernel_primitive:NN \Umathsubshiftdown              \utex_subshiftdown:D
1091   \__kernel_primitive:NN \Umathsubshiftdrop              \utex_subshiftdrop:D
1092   \__kernel_primitive:NN \Umathsubsupshiftdown           \utex_subsupshiftdown:D
1093   \__kernel_primitive:NN \Umathsubsupvgap                \utex_subsupvgap:D
1094   \__kernel_primitive:NN \Umathsubtopmax                 \utex_subtopmax:D
1095   \__kernel_primitive:NN \Umathsupbottommin              \utex_supbottommin:D
1096   \__kernel_primitive:NN \Umathsupshiftdrop              \utex_supshiftdrop:D
1097   \__kernel_primitive:NN \Umathsupshiftup                \utex_supshiftup:D
1098   \__kernel_primitive:NN \Umathsupsubbottommax           \utex_supsubbottommax:D
1099   \__kernel_primitive:NN \Umathunderbarkern              \utex_underbarkern:D
1100   \__kernel_primitive:NN \Umathunderbarrule              \utex_underbarrule:D
1101   \__kernel_primitive:NN \Umathunderbarvgap              \utex_underbarvgap:D
1102   \__kernel_primitive:NN \Umathunderdelimiterbgap        \utex_underdelimiterbgap:D
1103   \__kernel_primitive:NN \Umathunderdelimitervgap        \utex_underdelimitervgap:D
1104   \__kernel_primitive:NN \Uoverdelimiter                 \utex_overdelimiter:D
1105   \__kernel_primitive:NN \Uradical                       \utex_radical:D
1106   \__kernel_primitive:NN \Uroot                          \utex_root:D
1107   \__kernel_primitive:NN \Uskewed                        \utex_skewed:D
1108   \__kernel_primitive:NN \Uskewedwithdelims               \utex_skewedwithdelims:D
1109   \__kernel_primitive:NN \Ustack                         \utex_stack:D
1110   \__kernel_primitive:NN \Ustartdisplaymath               \utex_startdisplaymath:D
1111   \__kernel_primitive:NN \Ustartmath                     \utex_startmath:D
1112   \__kernel_primitive:NN \Ustopdisplaymath               \utex_stopdisplaymath:D
1113   \__kernel_primitive:NN \Ustopmath                      \utex_stopmath:D
1114   \__kernel_primitive:NN \Usubscript                     \utex_subscript:D
1115   \__kernel_primitive:NN \Usuperscript                   \utex_superscript:D
1116   \__kernel_primitive:NN \Uunderdelimiter                \utex_underdelimiter:D
1117   \__kernel_primitive:NN \Uvextensible                   \utex_vextensible:D
```

Primitives from pTeX.

```
1118   \__kernel_primitive:NN \autospacing                    \ptex_autospacing:D
1119   \__kernel_primitive:NN \autoxspacing                   \ptex_autoxspacing:D
1120   \__kernel_primitive:NN \dtou                           \ptex_dtou:D
1121   \__kernel_primitive:NN \euc                            \ptex_euc:D
1122   \__kernel_primitive:NN \ifdbox                         \ptex_ifdbox:D
```

246

```
1123    \__kernel_primitive:NN \ifddir                  \ptex_ifddir:D
1124    \__kernel_primitive:NN \ifmdir                  \ptex_ifmdir:D
1125    \__kernel_primitive:NN \iftbox                  \ptex_iftbox:D
1126    \__kernel_primitive:NN \iftdir                  \ptex_iftdir:D
1127    \__kernel_primitive:NN \ifybox                  \ptex_ifybox:D
1128    \__kernel_primitive:NN \ifydir                  \ptex_ifydir:D
1129    \__kernel_primitive:NN \inhibitglue             \ptex_inhibitglue:D
1130    \__kernel_primitive:NN \inhibitxspcode          \ptex_inhibitxspcode:D
1131    \__kernel_primitive:NN \jcharwidowpenalty       \ptex_jcharwidowpenalty:D
1132    \__kernel_primitive:NN \jfam                    \ptex_jfam:D
1133    \__kernel_primitive:NN \jfont                   \ptex_jfont:D
1134    \__kernel_primitive:NN \jis                     \ptex_jis:D
1135    \__kernel_primitive:NN \kanjiskip               \ptex_kanjiskip:D
1136    \__kernel_primitive:NN \kansuji                 \ptex_kansuji:D
1137    \__kernel_primitive:NN \kansujichar             \ptex_kansujichar:D
1138    \__kernel_primitive:NN \kcatcode                \ptex_kcatcode:D
1139    \__kernel_primitive:NN \kuten                   \ptex_kuten:D
1140    \__kernel_primitive:NN \noautospacing           \ptex_noautospacing:D
1141    \__kernel_primitive:NN \noautoxspacing          \ptex_noautoxspacing:D
1142    \__kernel_primitive:NN \postbreakpenalty        \ptex_postbreakpenalty:D
1143    \__kernel_primitive:NN \prebreakpenalty         \ptex_prebreakpenalty:D
1144    \__kernel_primitive:NN \showmode                \ptex_showmode:D
1145    \__kernel_primitive:NN \sjis                    \ptex_sjis:D
1146    \__kernel_primitive:NN \tate                    \ptex_tate:D
1147    \__kernel_primitive:NN \tbaselineshift          \ptex_tbaselineshift:D
1148    \__kernel_primitive:NN \tfont                   \ptex_tfont:D
1149    \__kernel_primitive:NN \xkanjiskip              \ptex_xkanjiskip:D
1150    \__kernel_primitive:NN \xspcode                 \ptex_xspcode:D
1151    \__kernel_primitive:NN \ybaselineshift          \ptex_ybaselineshift:D
1152    \__kernel_primitive:NN \yoko                    \ptex_yoko:D
```

Primitives from upTeX.

```
1153    \__kernel_primitive:NN \disablecjktoken         \uptex_disablecjktoken:D
1154    \__kernel_primitive:NN \enablecjktoken          \uptex_enablecjktoken:D
1155    \__kernel_primitive:NN \forcecjktoken           \uptex_forcecjktoken:D
1156    \__kernel_primitive:NN \kchar                   \uptex_kchar:D
1157    \__kernel_primitive:NN \kchardef                \uptex_kchardef:D
1158    \__kernel_primitive:NN \kuten                   \uptex_kuten:D
1159    \__kernel_primitive:NN \ucs                     \uptex_ucs:D
```

End of the "just the names" part of the source.

```
1160    ⟨/initex | names | package⟩
1161    ⟨*initex | package⟩
```

The job is done: close the group (using the primitive renamed!).

```
1162    \tex_endgroup:D
```

LaTeX 2ε will have moved a few primitives, so these are sorted out. A convenient test for LaTeX 2ε is the \@@end saved primitive.

```
1163    ⟨*package⟩
1164    \etex_ifdefined:D \@@end
1165      \tex_let:D \tex_end:D                    \@@end
1166      \tex_let:D \tex_everydisplay:D           \frozen@everydisplay
1167      \tex_let:D \tex_everymath:D              \frozen@everymath
1168      \tex_let:D \tex_hyphen:D                 \@@hyph
1169      \tex_let:D \tex_input:D                  \@@input
```

247

```
1170    \tex_let:D \tex_italiccorrection:D       \@@italiccorr
1171    \tex_let:D \tex_underline:D              \@@underline
```

Some tidying up is needed for \(pdf)tracingfonts. Newer LuaTeX has this simply as \tracingfonts, but that will have been 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.)

```
1172    \tex_let:D \pdftex_tracingfonts:D \tex_undefined:D
1173    \etex_ifdefined:D \pdftracingfonts
1174      \tex_let:D \pdftex_tracingfonts:D \pdftracingfonts
1175    \tex_else:D
1176      \etex_ifdefined:D \luatex_directlua:D
1177        \luatex_directlua:D { tex.enableprimitives("@@", {"tracingfonts"}) }
1178        \tex_let:D \pdftex_tracingfonts:D \luatextracingfonts
1179      \tex_fi:D
1180    \tex_fi:D
1181  \tex_fi:D
```

That is also true for the LuaTeX primitives under LaTeX 2ε (depending on the format-building date). There are a few primitives that get the right names anyway so are missing here!

```
1182  \etex_ifdefined:D \luatexsuppressfontnotfounderror
1183    \tex_let:D \luatex_alignmark:D                \luatexalignmark
1184    \tex_let:D \luatex_aligntab:D                 \luatexaligntab
1185    \tex_let:D \luatex_attribute:D                \luatexattribute
1186    \tex_let:D \luatex_attributedef:D             \luatexattributedef
1187    \tex_let:D \luatex_catcodetable:D             \luatexcatcodetable
1188    \tex_let:D \luatex_clearmarks:D               \luatexclearmarks
1189    \tex_let:D \luatex_crampeddisplaystyle:D      \luatexcrampeddisplaystyle
1190    \tex_let:D \luatex_crampedscriptscriptstyle:D \luatexcrampedscriptscriptstyle
1191    \tex_let:D \luatex_crampedscriptstyle:D       \luatexcrampedscriptstyle
1192    \tex_let:D \luatex_crampedtextstyle:D         \luatexcrampedtextstyle
1193    \tex_let:D \luatex_fontid:D                   \luatexfontid
1194    \tex_let:D \luatex_formatname:D               \luatexformatname
1195    \tex_let:D \luatex_gleaders:D                 \luatexgleaders
1196    \tex_let:D \luatex_initcatcodetable:D         \luatexinitcatcodetable
1197    \tex_let:D \luatex_latelua:D                  \luatexlatelua
1198    \tex_let:D \luatex_luaescapestring:D          \luatexluaescapestring
1199    \tex_let:D \luatex_luafunction:D              \luatexluafunction
1200    \tex_let:D \luatex_mathstyle:D                \luatexmathstyle
1201    \tex_let:D \luatex_nokerns:D                  \luatexnokerns
1202    \tex_let:D \luatex_noligs:D                   \luatexnoligs
1203    \tex_let:D \luatex_outputbox:D                \luatexoutputbox
1204    \tex_let:D \luatex_pageleftoffset:D           \luatexpageleftoffset
1205    \tex_let:D \luatex_pagetopoffset:D            \luatexpagetopoffset
1206    \tex_let:D \luatex_postexhyphenchar:D         \luatexpostexhyphenchar
1207    \tex_let:D \luatex_posthyphenchar:D           \luatexposthyphenchar
1208    \tex_let:D \luatex_preexhyphenchar:D          \luatexpreexhyphenchar
1209    \tex_let:D \luatex_prehyphenchar:D            \luatexprehyphenchar
1210    \tex_let:D \luatex_savecatcodetable:D         \luatexsavecatcodetable
1211    \tex_let:D \luatex_scantextokens:D            \luatexscantextokens
1212    \tex_let:D \luatex_suppressifcsnameerror:D    \luatexsuppressifcsnameerror
```

```
1213    \tex_let:D \luatex_suppresslongerror:D          \luatexsuppresslongerror
1214    \tex_let:D \luatex_suppressmathparerror:D       \luatexsuppressmathparerror
1215    \tex_let:D \luatex_suppressoutererror:D         \luatexsuppressoutererror
1216    \tex_let:D \utex_char:D                         \luatexUchar
1217    \tex_let:D \xetex_suppressfontnotfounderror:D   \luatexsuppressfontnotfounderror
```
Which also covers those slightly odd ones.
```
1218    \tex_let:D \luatex_bodydir:D               \luatexbodydir
1219    \tex_let:D \luatex_boxdir:D                \luatexboxdir
1220    \tex_let:D \luatex_leftghost:D             \luatexleftghost
1221    \tex_let:D \luatex_localbrokenpenalty:D    \luatexlocalbrokenpenalty
1222    \tex_let:D \luatex_localinterlinepenalty:D \luatexlocalinterlinepenalty
1223    \tex_let:D \luatex_localleftbox:D          \luatexlocalleftbox
1224    \tex_let:D \luatex_localrightbox:D         \luatexlocalrightbox
1225    \tex_let:D \luatex_mathdir:D               \luatexmathdir
1226    \tex_let:D \luatex_pagebottomoffset:D      \luatexpagebottomoffset
1227    \tex_let:D \luatex_pagedir:D               \luatexpagedir
1228    \tex_let:D \pdftex_pageheight:D            \luatexpageheight
1229    \tex_let:D \luatex_pagerightoffset:D       \luatexpagerightoffset
1230    \tex_let:D \pdftex_pagewidth:D             \luatexpagewidth
1231    \tex_let:D \luatex_pardir:D                \luatexpardir
1232    \tex_let:D \luatex_rightghost:D            \luatexrightghost
1233    \tex_let:D \luatex_textdir:D               \luatextextdir
1234  \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!
```
1235  \tex_ifnum:D 0
1236    \etex_ifdefined:D \pdftex_pdftexversion:D 1 \tex_fi:D
1237    \etex_ifdefined:D \luatex_luatexversion:D 1 \tex_fi:D
1238      = 0 %
1239    \tex_let:D \pdftex_mapfile:D \tex_undefined:D
1240    \tex_let:D \pdftex_mapline:D \tex_undefined:D
1241  \tex_fi:D
1242  ⟨/package⟩
```
Older XƎTEX versions use \XeTeX as the prefix for the Unicode math primitives it knows. That is tided up here (we support XƎTEX versions from 0.9994 but this change was in 0.9999).
```
1243  ⟨*initex | package⟩
1244  \etex_ifdefined:D \XeTeXdelcode
1245    \tex_let:D \utex_delcode:D        \XeTeXdelcode
1246    \tex_let:D \utex_delcodenum:D     \XeTeXdelcodenum
1247    \tex_let:D \utex_delimiter:D      \XeTeXdelimiter
1248    \tex_let:D \utex_mathaccent:D     \XeTeXmathaccent
1249    \tex_let:D \utex_mathchar:D       \XeTeXmathchar
1250    \tex_let:D \utex_mathchardef:D    \XeTeXmathchardef
1251    \tex_let:D \utex_mathcharnum:D    \XeTeXmathcharnum
1252    \tex_let:D \utex_mathcharnumdef:D \XeTeXmathcharnumdef
1253    \tex_let:D \utex_mathcode:D       \XeTeXmathcode
1254    \tex_let:D \utex_mathcodenum:D    \XeTeXmathcodenum
1255  \tex_fi:D
```
Up to v0.80, LuaTeX defines the pdfTeX version data: rather confusing. Removing them means that \pdftex_pdftexversion:D is a marker for pdfTeX alone: useful in engine-dependent code later.

```
1256  \etex_ifdefined:D \luatex_luatexversion:D
1257    \tex_let:D \pdftex_pdftexbanner:D    \tex_undefined:D
1258    \tex_let:D \pdftex_pdftexrevision:D \tex_undefined:D
1259    \tex_let:D \pdftex_pdftexversion:D  \tex_undefined:D
1260  \tex_fi:D
1261  ⟨/initex | package⟩
```

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.

```
1262  ⟨*package⟩
1263  \etex_ifdefined:D \normalend
1264    \tex_let:D \tex_end:D          \normalend
1265    \tex_let:D \tex_everyjob:D     \normaleveryjob
1266    \tex_let:D \tex_input:D        \normalinput
1267    \tex_let:D \tex_language:D     \normallanguage
1268    \tex_let:D \tex_mathop:D       \normalmathop
1269    \tex_let:D \tex_month:D        \normalmonth
1270    \tex_let:D \tex_outer:D        \normalouter
1271    \tex_let:D \tex_over:D         \normalover
1272    \tex_let:D \tex_vcenter:D      \normalvcenter
1273    \tex_let:D \etex_unexpanded:D  \normalunexpanded
1274    \tex_let:D \luatex_expanded:D  \normalexpanded
1275  \tex_fi:D
1276  \etex_ifdefined:D \normalitaliccorrection
1277    \tex_let:D \tex_hoffset:D           \normalhoffset
1278    \tex_let:D \tex_italiccorrection:D  \normalitaliccorrection
1279    \tex_let:D \tex_voffset:D           \normalvoffset
1280    \tex_let:D \etex_showtokens:D       \normalshowtokens
1281    \tex_let:D \luatex_bodydir:D        \spac_directions_normal_body_dir
1282    \tex_let:D \luatex_pagedir:D        \spac_directions_normal_page_dir
1283  \tex_fi:D
1284  \etex_ifdefined:D \normalleft
1285    \tex_let:D \tex_left:D    \normalleft
1286    \tex_let:D \tex_middle:D  \normalmiddle
1287    \tex_let:D \tex_right:D   \normalright
1288  \tex_fi:D
1289  ⟨/package⟩
1290  ⟨/initex | package⟩
```

3   l3basics implementation

```
1291  ⟨*initex | package⟩
```

3.1   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 do a few now, just to get started.[5]

---

[5]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.

`\if_true:` Then some conditionals.
`\if_false:`
`\or:`
`\else:`
`\fi:`
`\reverse_if:N`
`\if:w`
`\if_charcode:w`
`\if_catcode:w`
`\if_meaning:w`

```
1292 \tex_let:D \if_true:           \tex_iftrue:D
1293 \tex_let:D \if_false:          \tex_iffalse:D
1294 \tex_let:D \or:                \tex_or:D
1295 \tex_let:D \else:              \tex_else:D
1296 \tex_let:D \fi:                \tex_fi:D
1297 \tex_let:D \reverse_if:N       \etex_unless:D
1298 \tex_let:D \if:w               \tex_if:D
1299 \tex_let:D \if_charcode:w       \tex_if:D
1300 \tex_let:D \if_catcode:w       \tex_ifcat:D
1301 \tex_let:D \if_meaning:w       \tex_ifx:D
```

(*End definition for* `\if_true:` *and others. These functions are documented on page 21.*)

`\if_mode_math:` TeX lets us detect some if its modes.
`\if_mode_horizontal:`
`\if_mode_vertical:`
`\if_mode_inner:`

```
1302 \tex_let:D \if_mode_math:       \tex_ifmmode:D
1303 \tex_let:D \if_mode_horizontal: \tex_ifhmode:D
1304 \tex_let:D \if_mode_vertical:   \tex_ifvmode:D
1305 \tex_let:D \if_mode_inner:      \tex_ifinner:D
```

(*End definition for* `\if_mode_math:` *and others. These functions are documented on page 21.*)

`\if_cs_exist:N` Building csnames and testing if control sequences exist.
`\if_cs_exist:w`
`\cs:w`
`\cs_end:`

```
1306 \tex_let:D \if_cs_exist:N       \etex_ifdefined:D
1307 \tex_let:D \if_cs_exist:w       \etex_ifcsname:D
1308 \tex_let:D \cs:w                \tex_csname:D
1309 \tex_let:D \cs_end:             \tex_endcsname:D
```

(*End definition for* `\if_cs_exist:N` *and others. These functions are documented on page 21.*)

`\exp_after:wN` The five `\exp_` functions are used in the l3expan module where they are described.
`\exp_not:N`
`\exp_not:n`

```
1310 \tex_let:D \exp_after:wN         \tex_expandafter:D
1311 \tex_let:D \exp_not:N            \tex_noexpand:D
1312 \tex_let:D \exp_not:n            \etex_unexpanded:D
1313 \tex_let:D \exp:w                \tex_romannumeral:D
1314 \tex_chardef:D \exp_end:  = 0 ~
```

(*End definition for* `\exp_after:wN`, `\exp_not:N`, *and* `\exp_not:n`. *These functions are documented on page 30.*)

`\token_to_meaning:N` Examining a control sequence or token.
`\cs_meaning:N`

```
1315 \tex_let:D \token_to_meaning:N \tex_meaning:D
1316 \tex_let:D \cs_meaning:N        \tex_meaning:D
```

(*End definition for* `\token_to_meaning:N` *and* `\cs_meaning:N`. *These functions are documented on page 52.*)

`\tl_to_str:n` Making strings.
`\token_to_str:N`

```
1317 \tex_let:D \tl_to_str:n         \etex_detokenize:D
1318 \tex_let:D \token_to_str:N      \tex_string:D
```

(*End definition for* `\tl_to_str:n` *and* `\token_to_str:N`. *These functions are documented on page 96.*)

`\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.

```
1319 \tex_let:D \scan_stop:          \tex_relax:D
1320 \tex_let:D \group_begin:        \tex_begingroup:D
1321 \tex_let:D \group_end:          \tex_endgroup:D
```

(*End definition for* `\scan_stop:` *,* `\group_begin:` *, and* `\group_end:`*. These functions are documented on page 9.*)

`\if_int_compare:w`
`\__int_to_roman:w`

For integers.

```
1322 \tex_let:D \if_int_compare:w    \tex_ifnum:D
1323 \tex_let:D \__int_to_roman:w    \tex_romannumeral:D
```

(*End definition for* `\if_int_compare:w` *and* `\__int_to_roman:w`*. These functions are documented on page 73.*)

`\group_insert_after:N`

Adding material after the end of a group.

```
1324 \tex_let:D \group_insert_after:N \tex_aftergroup:D
```

(*End definition for* `\group_insert_after:N`*. This function is documented on page 9.*)

`\exp_args:Nc`
`\exp_args:cc`

Discussed in l3expan, but needed much earlier.

```
1325 \tex_long:D \tex_def:D \exp_args:Nc #1#2
1326   { \exp_after:wN #1 \cs:w #2 \cs_end: }
1327 \tex_long:D \tex_def:D \exp_args:cc #1#2
1328   { \cs:w #1 \exp_after:wN \cs_end: \cs:w #2 \cs_end: }
```

(*End definition for* `\exp_args:Nc` *and* `\exp_args:cc`*. These functions are documented on page 27.*)

`\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.

```
1329 \tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
1330 \tex_long:D \tex_def:D \cs_meaning:c #1
1331   {
1332     \if_cs_exist:w #1 \cs_end:
1333       \exp_after:wN \use_i:nn
1334     \else:
1335       \exp_after:wN \use_ii:nn
1336     \fi:
1337     { \exp_args:Nc \cs_meaning:N {#1} }
1338     { \tl_to_str:n {undefined} }
1339   }
1340 \tex_let:D \token_to_meaning:c = \cs_meaning:c
```

(*End definition for* `\token_to_meaning:c` *,* `\token_to_str:c` *, and* `\cs_meaning:c`*. These functions are documented on page 52.*)

3.2 Defining some constants

\c_minus_one
\c_zero
\c_sixteen
We need the constants `\c_minus_one` and `\c_sixteen` now for writing information to the log and the terminal and `\c_zero` which is used by some functions in the l3alloc 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! The actual allocation mechanism is in l3alloc, and works such that the first available count register is 10.

```
1341 ⟨*package⟩
1342 \tex_let:D \c_minus_one \m@ne
1343 ⟨/package⟩
1344 ⟨*initex⟩
1345 \tex_countdef:D \c_minus_one = 10 ~
1346 \c_minus_one = -1 ~
1347 ⟨/initex⟩
1348 \tex_chardef:D \c_sixteen = 16 ~
1349 \tex_chardef:D \c_zero    = 0 ~
```

(*End definition for* `\c_minus_one`, `\c_zero`, *and* `\c_sixteen`. *These variables are documented on page* *72.*)

\c_max_register_int
This is here as this particular integer is needed both in package mode and to bootstrap l3alloc, and is documented in l3int.

```
1350 \etex_ifdefined:D \luatex_luatexversion:D
1351   \tex_chardef:D \c_max_register_int = 65 535 ~
1352 \tex_else:D
1353   \tex_mathchardef:D \c_max_register_int = 32 767 ~
1354 \tex_fi:D
```

(*End definition for* `\c_max_register_int`. *This variable is documented on page* *72.*)

3.3 Defining functions

We start by providing functions for the typical definition functions. First the local ones.

\cs_set_nopar:Npn
\cs_set_nopar:Npx
\cs_set:Npn
\cs_set:Npx
\cs_set_protected_nopar:Npn
\cs_set_protected_nopar:Npx
\cs_set_protected:Npn
\cs_set_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.

```
1355 \tex_let:D \cs_set_nopar:Npn            \tex_def:D
1356 \tex_let:D \cs_set_nopar:Npx            \tex_edef:D
1357 \etex_protected:D \tex_long:D \tex_def:D \cs_set:Npn
1358   { \tex_long:D \tex_def:D }
1359 \etex_protected:D \tex_long:D \tex_def:D \cs_set:Npx
1360   { \tex_long:D \tex_edef:D }
1361 \etex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npn
1362   { \etex_protected:D \tex_def:D }
1363 \etex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npx
1364   { \etex_protected:D \tex_edef:D }
1365 \etex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npn
1366   { \etex_protected:D \tex_long:D \tex_def:D }
1367 \etex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npx
1368   { \etex_protected:D \tex_long:D \tex_edef:D }
```

(*End definition for* `\cs_set_nopar:Npn` *and others. These functions are documented on page* *11.*)

\cs_gset_nopar:Npn
\cs_gset_nopar:Npx
\cs_gset:Npn
\cs_gset:Npx
\cs_gset_protected_nopar:Npn
\cs_gset_protected_nopar:Npx
\cs_gset_protected:Npn
\cs_gset_protected:Npx

Global versions of the above functions.

```
1369 \tex_let:D \cs_gset_nopar:Npn              \tex_gdef:D
1370 \tex_let:D \cs_gset_nopar:Npx              \tex_xdef:D
1371 \cs_set_protected:Npn \cs_gset:Npn
1372   { \tex_long:D \tex_gdef:D }
1373 \cs_set_protected:Npn \cs_gset:Npx
1374   { \tex_long:D \tex_xdef:D }
1375 \cs_set_protected:Npn \cs_gset_protected_nopar:Npn
1376   { \etex_protected:D \tex_gdef:D }
1377 \cs_set_protected:Npn \cs_gset_protected_nopar:Npx
1378   { \etex_protected:D \tex_xdef:D }
1379 \cs_set_protected:Npn \cs_gset_protected:Npn
1380   { \etex_protected:D \tex_long:D \tex_gdef:D }
1381 \cs_set_protected:Npn \cs_gset_protected:Npx
1382   { \etex_protected:D \tex_long:D \tex_xdef:D }
```

(*End definition for* \cs_gset_nopar:Npn *and others. These functions are documented on page 12.*)

3.4   Selecting tokens

\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.

```
1383 \cs_set_nopar:Npn \l__exp_internal_tl { }
```

(*End definition for* \l__exp_internal_tl.)

\use:c   This macro grabs its argument and returns a csname from it.

```
1384 \cs_set:Npn \use:c #1 { \cs:w #1 \cs_end: }
```

(*End definition for* \use:c. *This function is documented on page 16.*)

\use:x   Fully expands its argument and passes it to the input stream. Uses the reserved \l__-exp_internal_tl which will be set up in l3expan.

```
1385 \cs_set_protected:Npn \use:x #1
1386   {
1387     \cs_set_nopar:Npx \l__exp_internal_tl {#1}
1388     \l__exp_internal_tl
1389   }
```

(*End definition for* \use:x. *This function is documented on page 18.*)

\use:n
\use:nn
\use:nnn
\use:nnnn

These macros grab their arguments and returns them back to the input (with outer braces removed).

```
1390 \cs_set:Npn \use:n    #1       {#1}
1391 \cs_set:Npn \use:nn   #1#2     {#1#2}
1392 \cs_set:Npn \use:nnn  #1#2#3   {#1#2#3}
1393 \cs_set:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}
```

(*End definition for* \use:n *and others. These functions are documented on page 17.*)

\use_i:nn
\use_ii:nn

The equivalent to LaTeX2ε's \@firstoftwo and \@secondoftwo.

```
1394 \cs_set:Npn \use_i:nn  #1#2 {#1}
1395 \cs_set:Npn \use_ii:nn #1#2 {#2}
```

(*End definition for* \use_i:nn *and* \use_ii:nn. *These functions are documented on page 17.*)

We also need something for picking up arguments from a longer list.

```
1396 \cs_set:Npn \use_i:nnn    #1#2#3 {#1}
1397 \cs_set:Npn \use_ii:nnn   #1#2#3 {#2}
1398 \cs_set:Npn \use_iii:nnn  #1#2#3 {#3}
1399 \cs_set:Npn \use_i_ii:nnn #1#2#3 {#1#2}
1400 \cs_set:Npn \use_i:nnnn   #1#2#3#4 {#1}
1401 \cs_set:Npn \use_ii:nnnn  #1#2#3#4 {#2}
1402 \cs_set:Npn \use_iii:nnnn #1#2#3#4 {#3}
1403 \cs_set:Npn \use_iv:nnnn  #1#2#3#4 {#4}
```

(*End definition for* \use_i:nnn *and others. These functions are documented on page 18.*)

Functions that gobble everything until they see either \q_nil, \q_stop, or \q_-recursion_stop, respectively.

```
1404 \cs_set:Npn \use_none_delimit_by_q_nil:w  #1 \q_nil  { }
1405 \cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
1406 \cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop { }
```

(*End 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 19.*)

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.

```
1407 \cs_set:Npn \use_i_delimit_by_q_nil:nw  #1#2 \q_nil  {#1}
1408 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
1409 \cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw #1#2 \q_recursion_stop {#1}
```

(*End 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 19.*)

3.5 Gobbling tokens from input

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 will take care of gobbling all the tokens in one go.

```
1410 \cs_set:Npn \use_none:n         #1                  { }
1411 \cs_set:Npn \use_none:nn        #1#2                { }
1412 \cs_set:Npn \use_none:nnn       #1#2#3              { }
1413 \cs_set:Npn \use_none:nnnn      #1#2#3#4            { }
1414 \cs_set:Npn \use_none:nnnnn     #1#2#3#4#5          { }
1415 \cs_set:Npn \use_none:nnnnnn    #1#2#3#4#5#6        { }
1416 \cs_set:Npn \use_none:nnnnnnn   #1#2#3#4#5#6#7      { }
1417 \cs_set:Npn \use_none:nnnnnnnn  #1#2#3#4#5#6#7#8    { }
1418 \cs_set:Npn \use_none:nnnnnnnnn #1#2#3#4#5#6#7#8#9 { }
```

(*End definition for* \use_none:n *and others. These functions are documented on page 18.*)

3.6 Conditional processing and definitions

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` will expand fully any `\else:` and the `\fi:` that are waiting to be discarded, before reaching the `\exp_end:` which will leave the expansion null. 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.

```
1419 \cs_set:Npn \prg_return_true:
1420   { \exp_after:wN \use_i:nn  \exp:w }
1421 \cs_set:Npn \prg_return_false:
1422   { \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 will work.

(*End definition for* `\prg_return_true:` *and* `\prg_return_false:`. *These functions are documented on page 36.*)

`\prg_set_conditional:Npnn`
`\prg_new_conditional:Npnn`
`\prg_set_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.

```
1423 \cs_set_protected:Npn \prg_set_conditional:Npnn
1424   { \__prg_generate_conditional_parm:nnNpnn { set } { } }
1425 \cs_set_protected:Npn \prg_new_conditional:Npnn
1426   { \__prg_generate_conditional_parm:nnNpnn { new } { } }
1427 \cs_set_protected:Npn \prg_set_protected_conditional:Npnn
1428   { \__prg_generate_conditional_parm:nnNpnn { set } { _protected } }
1429 \cs_set_protected:Npn \prg_new_protected_conditional:Npnn
1430   { \__prg_generate_conditional_parm:nnNpnn { new } { _protected } }
1431 \cs_set_protected:Npn \__prg_generate_conditional_parm:nnNpnn #1#2#3#4#
```

```
1432    {
1433      \__cs_split_function:NN #3 \__prg_generate_conditional:nnNnnnnn
1434      {#1} {#2} {#4}
1435    }
```

(*End definition for* `\prg_set_conditional:Npnn` *and others. These functions are documented on page* *34.*)

`\prg_set_conditional:Nnn`
`\prg_new_conditional:Nnn`
`\prg_set_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.

```
1436  \cs_set_protected:Npn \prg_set_conditional:Nnn
1437    { \__prg_generate_conditional_count:nnNnn { set } { } }
1438  \cs_set_protected:Npn \prg_new_conditional:Nnn
1439    { \__prg_generate_conditional_count:nnNnn { new } { } }
1440  \cs_set_protected:Npn \prg_set_protected_conditional:Nnn
1441    { \__prg_generate_conditional_count:nnNnn { set } { _protected } }
1442  \cs_set_protected:Npn \prg_new_protected_conditional:Nnn
1443    { \__prg_generate_conditional_count:nnNnn { new } { _protected } }
1444  \cs_set_protected:Npn \__prg_generate_conditional_count:nnNnn #1#2#3
1445    {
1446      \__cs_split_function:NN #3 \__prg_generate_conditional_count:nnNnnnn
1447      {#1} {#2}
1448    }
1449  \cs_set_protected:Npn \__prg_generate_conditional_count:nnNnnnn #1#2#3#4#5
1450    {
1451      \__cs_parm_from_arg_count:nnF
1452        { \__prg_generate_conditional:nnNnnnnn {#1} {#2} #3 {#4} {#5} }
1453        { \tl_count:n {#2} }
1454        {
1455          \__msg_kernel_error:nnxx { kernel } { bad-number-of-arguments }
1456            { \token_to_str:c { #1 : #2 } }
1457            { \tl_count:n {#2} }
1458          \use_none:nn
1459        }
1460    }
```

(*End definition for* `\prg_set_conditional:Nnn` *and others. These functions are documented on page* *34.*)

`\__prg_generate_conditional:nnNnnnnn`
`\__prg_generate_conditional:nnnnnnw`

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.

```
1461  \cs_set_protected:Npn \__prg_generate_conditional:nnNnnnnn #1#2#3#4#5#6#7#8
1462    {
1463      \if_meaning:w \c_false_bool #3
1464        \__msg_kernel_error:nnx { kernel } { missing-colon }
1465          { \token_to_str:c {#1} }
1466        \exp_after:wN \use_none:nn
1467      \fi:
1468      \use:x
1469        {
1470          \exp_not:N \__prg_generate_conditional:nnnnnnw
1471          \exp_not:n { {#4} {#5} {#1} {#2} {#6} {#8} }
1472          \tl_to_str:n {#7}
1473          \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
1474        }
1475    }
```

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:nnnnnnn` cleans up. Otherwise, the error message is removed by the variant form.

```
1476  \cs_set_protected:Npn \__prg_generate_conditional:nnnnnnw #1#2#3#4#5#6#7 ,
1477    {
1478      \if_meaning:w \q_recursion_tail #7
1479        \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1480      \fi:
1481      \use:c { __prg_generate_ #7 _form:wnnnnnn }
1482        \tl_if_empty:nF {#7}
1483          {
1484            \__msg_kernel_error:nnxx
1485              { kernel } { conditional-form-unknown }
1486              {#7} { \token_to_str:c { #3 : #4 } }
1487          }
1488        \use_none:nnnnnnn
1489      \q_stop
1490      {#1} {#2} {#3} {#4} {#5} {#6}
1491      \__prg_generate_conditional:nnnnnnw {#1} {#2} {#3} {#4} {#5} {#6}
1492    }
```

(*End definition for* `\__prg_generate_conditional:nnNnnnnn` *and* `\__prg_generate_conditional:nnnnnnw`.)

`\__prg_generate_p_form:wnnnnnn`
`\__prg_generate_TF_form:wnnnnnn`
`\__prg_generate_T_form:wnnnnnn`
`\__prg_generate_F_form:wnnnnnn`

How to generate the various forms. Those functions take the following arguments: 1: `set` or `new`, 2: empty or `_protected`, 3: function name 4: signature, 5: parameter text (or empty), 6: replacement. 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` variant 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.

```
1493  \cs_set_protected:Npn \__prg_generate_p_form:wnnnnnn
1494    #1 \q_stop #2#3#4#5#6#7
1495    {
1496      \if_meaning:w \scan_stop: #3 \scan_stop:
1497        \exp_after:wN \use_i:nn
1498      \else:
```

```
1499        \exp_after:wN \use_ii:nn
1500      \fi:
1501        {
1502        \exp_args:cc { cs_ #2 #3 :Npn } { #4 _p: #5 } #6
1503          { #7 \exp_end: \c_true_bool \c_false_bool }
1504      }
1505      {
1506        \__msg_kernel_error:nnx { kernel } { protected-predicate }
1507          { \token_to_str:c { #4 _p: #5 } }
1508      }
1509  }
1510 \cs_set_protected:Npn \__prg_generate_T_form:wnnnnnn
1511    #1 \q_stop #2#3#4#5#6#7
1512  {
1513    \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 T } #6
1514      { #7 \exp_end: \use:n \use_none:n }
1515  }
1516 \cs_set_protected:Npn \__prg_generate_F_form:wnnnnnn
1517    #1 \q_stop #2#3#4#5#6#7
1518  {
1519    \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 F } #6
1520      { #7 \exp_end: { } }
1521  }
1522 \cs_set_protected:Npn \__prg_generate_TF_form:wnnnnnn
1523    #1 \q_stop #2#3#4#5#6#7
1524  {
1525    \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 TF } #6
1526      { #7 \exp_end: }
1527  }
```

(*End definition for* __prg_generate_p_form:wnnnnnn *and others.*)

\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn
__prg_set_eq_conditional:NNNn

The setting-equal functions. Split both functions and feed $\{\langle name_1\rangle\}$ $\{\langle signature_1\rangle\}$ $\langle boolean_1\rangle$ $\{\langle name_2\rangle\}$ $\{\langle signature_2\rangle\}$ $\langle boolean_2\rangle$ $\langle copying\ function\rangle$ $\langle conditions\rangle$, \q_-recursion_tail , \q_recursion_stop to a first auxiliary.

```
1528 \cs_set_protected:Npn \prg_set_eq_conditional:NNn
1529    { \__prg_set_eq_conditional:NNNn \cs_set_eq:cc }
1530 \cs_set_protected:Npn \prg_new_eq_conditional:NNn
1531    { \__prg_set_eq_conditional:NNNn \cs_new_eq:cc }
1532 \cs_set_protected:Npn \__prg_set_eq_conditional:NNNn #1#2#3#4
1533  {
1534    \use:x
1535      {
1536        \exp_not:N \__prg_set_eq_conditional:nnNnnNNw
1537          \__cs_split_function:NN #2 \prg_do_nothing:
1538          \__cs_split_function:NN #3 \prg_do_nothing:
1539          \exp_not:N #1
1540          \tl_to_str:n {#4}
1541          \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
1542      }
1543  }
```

(*End definition for* \prg_set_eq_conditional:NNn, \prg_new_eq_conditional:NNn, *and* __prg_set_-eq_conditional:NNNn. *These functions are documented on page 35.*)

259

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 {⟨*name₁*⟩} {⟨*signature₁*⟩} {⟨*name₂*⟩} {⟨*signature₂*⟩} ⟨*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.

```
1544 \cs_set_protected:Npn \__prg_set_eq_conditional:nnNnnNNw #1#2#3#4#5#6
1545   {
1546     \if_meaning:w \c_false_bool #3
1547       \__msg_kernel_error:nnx { kernel } { missing-colon }
1548         { \token_to_str:c {#1} }
1549       \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1550     \fi:
1551     \if_meaning:w \c_false_bool #6
1552       \__msg_kernel_error:nnx { kernel } { missing-colon }
1553         { \token_to_str:c {#4} }
1554       \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1555     \fi:
1556     \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#4} {#5}
1557   }
1558 \cs_set_protected:Npn \__prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 ,
1559   {
1560     \if_meaning:w \q_recursion_tail #6
1561       \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
1562     \fi:
1563     \use:c { __prg_set_eq_conditional_ #6 _form:wNnnnn }
1564       \tl_if_empty:nF {#6}
1565         {
1566           \__msg_kernel_error:nnxx
1567             { kernel } { conditional-form-unknown }
1568             {#6} { \token_to_str:c { #1 : #2 } }
1569         }
1570       \use_none:nnnnnn
1571     \q_stop
1572     #5 {#1} {#2} {#3} {#4}
1573     \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
1574   }
1575 \cs_set:Npn \__prg_set_eq_conditional_p_form:wNnnnn #1 \q_stop #2#3#4#5#6
1576   {
1577     \__chk_if_exist_cs:c { #5 _p : #6     }
1578     #2 { #3 _p : #4     } { #5 _p : #6     }
1579   }
1580 \cs_set:Npn \__prg_set_eq_conditional_TF_form:wNnnnn #1 \q_stop #2#3#4#5#6
1581   {
1582     \__chk_if_exist_cs:c { #5    : #6 TF }
1583     #2 { #3    : #4 TF } { #5    : #6 TF }
1584   }
1585 \cs_set:Npn \__prg_set_eq_conditional_T_form:wNnnnn #1 \q_stop #2#3#4#5#6
1586   {
1587     \__chk_if_exist_cs:c { #5    : #6 T  }
1588     #2 { #3    : #4 T  } { #5    : #6 T  }
1589   }
```

260

```
1590 \cs_set:Npn \__prg_set_eq_conditional_F_form:wNnnnn #1 \q_stop #2#3#4#5#6
1591   {
1592     \__chk_if_exist_cs:c { #5    : #6  F }
1593     #2 { #3    : #4  F } { #5    : #6  F }
1594   }
```

(*End 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.

<code>\c_true_bool</code>
<code>\c_false_bool</code>
Here are the canonical boolean values.

```
1595 \tex_chardef:D \c_true_bool  = 1 ~
1596 \tex_chardef:D \c_false_bool = 0 ~
```

(*End definition for* `\c_true_bool` *and* `\c_false_bool`. *These variables are documented on page 20.*)

3.7   Dissecting a control sequence

<code>\cs_to_str:N</code>
<code>__cs_to_str:N</code>
<code>__cs_to_str:w</code>
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`. 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` 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.

```
1597 \cs_set:Npn \cs_to_str:N
1598   {
```

We implement the expansion scheme using `\tex_romannumeral:D` terminating it with `\c_zero` 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` so we make this dependency explicit.

```
1599       \tex_romannumeral:D
1600         \if:w \token_to_str:N \ \__cs_to_str:w \fi:
1601         \exp_after:wN \__cs_to_str:N \token_to_str:N
1602   }
1603 \cs_set:Npn \__cs_to_str:N #1 { \c_zero }
1604 \cs_set:Npn \__cs_to_str:w #1 \__cs_to_str:N
1605   { - \__int_value:w \fi: \exp_after:wN \c_zero }
```

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 definition for* `\cs_to_str:N`, `\__cs_to_str:N`, *and* `\__cs_to_str:w`. *These functions are documented on page 17.*)

`\__cs_split_function:NN`
`\__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. Lastly, the second argument of `\__cs_split_function:NN` is supposed to be a function taking three variables, one for name, one for signature, and one for the boolean. For example, `\__cs_split_function:NN \foo_bar:cnx \use_i:nnn` as input becomes `\use_i:nnn {foo_bar} {cnx} \c_true_bool`.

We cannot use : directly as it has the wrong category code so an x-type expansion is used to force the conversion.

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 `\q_mark`, then `\c_true_bool` as #3, and #4 cleans up until `\q_stop`. Otherwise, the #1 contains the function name and `\q_mark \c_true_bool`, #2 is empty, #3 is `\c_false_bool`, and #4 cleans up. In both cases, #5 is the ⟨*processor*⟩. The second auxiliary trims the trailing `\q_mark` from the function name if present (that is, if the original function had no colon).

```
1606 \cs_set:Npx \__cs_split_function:NN #1
1607   {
1608     \exp_not:N \exp_after:wN \exp_not:N \exp_after:wN
1609     \exp_not:N \exp_after:wN \exp_not:N \__cs_split_function_auxi:w
1610       \exp_not:N \cs_to_str:N #1 \exp_not:N \q_mark \c_true_bool
1611       \token_to_str:N : \exp_not:N \q_mark \c_false_bool
1612       \exp_not:N \q_stop
1613   }
1614 \use:x
1615   {
1616     \cs_set:Npn \exp_not:N \__cs_split_function_auxi:w
1617       ##1 \token_to_str:N : ##2 \exp_not:N \q_mark ##3##4 \exp_not:N \q_stop ##5
1618   }
1619   { \__cs_split_function_auxii:w #5 #1 \q_mark \q_stop {#2} #3 }
1620 \cs_set:Npn \__cs_split_function_auxii:w #1#2 \q_mark #3 \q_stop
1621   { #1 {#2} }
```

(*End definition for* __cs_split_function:NN, __cs_split_function_auxi:w, *and* __cs_split_-
function_auxii:w.)

__cs_get_function_name:N   Simple wrappers.
_cs_get_function_signature:N

```
1622 \cs_set:Npn \__cs_get_function_name:N #1
1623   { \__cs_split_function:NN #1 \use_i:nnn }
1624 \cs_set:Npn \__cs_get_function_signature:N #1
1625   { \__cs_split_function:NN #1 \use_ii:nnn }
```

(*End definition for* __cs_get_function_name:N *and* __cs_get_function_signature:N.)

3.8   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   Two versions for checking existence. For the N form we firstly check for \scan_stop: and
\cs_if_exist_p:c   then if it is in the hash table. There is no problem when inputting something like \else:
\cs_if_exist:N*TF*   or \fi: as TeX will only ever skip input in case the token tested against is \scan_stop:.
\cs_if_exist:c*TF*
```
1626 \prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
1627   {
1628     \if_meaning:w #1 \scan_stop:
1629       \prg_return_false:
1630     \else:
1631       \if_cs_exist:N #1
1632         \prg_return_true:
1633       \else:
1634         \prg_return_false:
1635       \fi:
1636     \fi:
1637   }
```
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.
```
1638 \prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
1639   {
1640     \if_cs_exist:w #1 \cs_end:
1641       \exp_after:wN \use_i:nn
1642     \else:
1643       \exp_after:wN \use_ii:nn
1644     \fi:
1645     {
1646       \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1647         \prg_return_false:
1648       \else:
1649         \prg_return_true:
1650       \fi:
1651     }
1652     \prg_return_false:
1653   }
```

263

\cs_if_free_p:N   The logical reversal of the above.
\cs_if_free_p:c
\cs_if_free:N*TF*
\cs_if_free:c*TF*

```
1654 \prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
1655   {
1656     \if_meaning:w #1 \scan_stop:
1657       \prg_return_true:
1658     \else:
1659       \if_cs_exist:N #1
1660         \prg_return_false:
1661       \else:
1662         \prg_return_true:
1663       \fi:
1664     \fi:
1665   }
1666 \prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
1667   {
1668     \if_cs_exist:w #1 \cs_end:
1669       \exp_after:wN \use_i:nn
1670     \else:
1671       \exp_after:wN \use_ii:nn
1672     \fi:
1673       {
1674         \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1675           \prg_return_true:
1676         \else:
1677           \prg_return_false:
1678         \fi:
1679       }
1680       { \prg_return_true: }
1681   }
```

\cs_if_exist_use:N   The \cs_if_exist_use:... functions cannot be implemented as conditionals because
\cs_if_exist_use:c   the true branch must leave both the control sequence itself and the true code in the input
\cs_if_exist_use:N*TF*   stream. For the c variants, we are careful not to put the control sequence in the hash
\cs_if_exist_use:c*TF*   table if it does not exist. In LuaTeX we could use the \lastnamedcs primitive.

```
1682 \cs_set:Npn \cs_if_exist_use:NTF #1#2
1683   { \cs_if_exist:NTF #1 { #1 #2 } }
1684 \cs_set:Npn \cs_if_exist_use:NF #1
1685   { \cs_if_exist:NTF #1 { #1 } }
1686 \cs_set:Npn \cs_if_exist_use:NT #1 #2
1687   { \cs_if_exist:NTF #1 { #1 #2 } { } }
1688 \cs_set:Npn \cs_if_exist_use:N #1
1689   { \cs_if_exist:NTF #1 { #1 } { } }
1690 \cs_set:Npn \cs_if_exist_use:cTF #1#2
1691   { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } }
1692 \cs_set:Npn \cs_if_exist_use:cF #1
1693   { \cs_if_exist:cTF {#1} { \use:c {#1} } }
1694 \cs_set:Npn \cs_if_exist_use:cT #1#2
1695   { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } }
1696 \cs_set:Npn \cs_if_exist_use:c #1
1697   { \cs_if_exist:cTF {#1} { \use:c {#1} } { } }
```

(End definition for `\cs_if_exist_use:NTF`*. This function is documented on page 16.)*

3.9 Defining and checking (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 definitions here are only temporary, they will be redefined later on.

`\iow_log:x`
`\iow_term:x`

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 l3io.

```
1698 \cs_set_protected:Npn \iow_log:x
1699   { \tex_immediate:D \tex_write:D \c_minus_one }
1700 \cs_set_protected:Npn \iow_term:x
1701   { \tex_immediate:D \tex_write:D \c_sixteen }
```

(End definition for `\iow_log:x` *and* `\iow_term:x`*. These functions are documented on page 175.)*

`\__chk_log:x`
`\__chk_suspend_log:`
`\__chk_resume_log:`

This function is used to write some information to the `log` file in case the `log-function` option is set. Otherwise its argument is ignored. Using this function rather than directly using `\iow_log:x` allows for `\__chk_suspend_log:` which disables such messages until the matching `\__chk_resume_log:`. These two commands are used to improve the logging for complicated datatypes. They should come in pairs, which can be nested. The function `\exp_not:o` is defined in l3expan later on but `\__chk_suspend_log:` and `\__chk_resume_log:` are not used before that point.

```
1702 ⟨*initex⟩
1703 \cs_set_protected:Npn \__chk_log:x { \use_none:n }
1704 \cs_set_protected:Npn \__chk_suspend_log: { }
1705 \cs_set_protected:Npn \__chk_resume_log: { }
1706 ⟨/initex⟩
1707 ⟨*package⟩
1708 \tex_ifodd:D \l@expl@log@functions@bool
1709   \cs_set_protected:Npn \__chk_log:x { \iow_log:x }
1710   \cs_set_protected:Npn \__chk_suspend_log:
1711     {
1712       \cs_set_protected:Npx \__chk_resume_log:
1713         {
1714           \cs_set_protected:Npn \__chk_resume_log:
1715             { \exp_not:o { \__chk_resume_log: } }
1716           \cs_set_protected:Npn \__chk_log:x
1717             { \exp_not:o { \__chk_log:x } }
1718         }
1719       \cs_set_protected:Npn \__chk_log:x { \use_none:n }
1720     }
1721   \cs_set_protected:Npn \__chk_resume_log: { }
1722 \else:
1723   \cs_set_protected:Npn \__chk_log:x { \use_none:n }
1724   \cs_set_protected:Npn \__chk_suspend_log: { }
1725   \cs_set_protected:Npn \__chk_resume_log: { }
1726 \fi:
1727 ⟨/package⟩
```

(End definition for `\__chk_log:x`*,* `\__chk_suspend_log:`*, and* `\__chk_resume_log:`*.)*

`\__msg_kernel_error:nnxx`
`\__msg_kernel_error:nnx`
`\__msg_kernel_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.

```
1728 \cs_set_protected:Npn \__msg_kernel_error:nnxx #1#2#3#4
1729   {
1730     \tex_newlinechar:D = '\^^J \tex_relax:D
1731     \tex_errmessage:D
1732       {
1733         !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^^J
1734         Argh,~internal~LaTeX3~error! ^^J ^^J
1735         Module ~ #1 , ~ message~name~"#2": ^^J
1736         Arguments~'#3'~and~'#4' ^^J ^^J
1737         This~is~one~for~The~LaTeX3~Project:~bailing~out
1738       }
1739     \tex_end:D
1740   }
1741 \cs_set_protected:Npn \__msg_kernel_error:nnx #1#2#3
1742   { \__msg_kernel_error:nnxx {#1} {#2} {#3} { } }
1743 \cs_set_protected:Npn \__msg_kernel_error:nn #1#2
1744   { \__msg_kernel_error:nnxx {#1} {#2} { } { } }
```

(End definition for `\__msg_kernel_error:nnxx`*,* `\__msg_kernel_error:nnx`*, and* `\__msg_kernel_error:nn`*.)*

`\msg_line_context:`

Another one from l3msg which will be altered later.

```
1745 \cs_set:Npn \msg_line_context:
1746   { on~line~ \tex_the:D \tex_inputlineno:D }
```

(End definition for `\msg_line_context:`*. This function is documented on page 151.)*

`\__chk_if_free_cs:N`
`\__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!

```
1747 \cs_set_protected:Npn \__chk_if_free_cs:N #1
1748   {
1749     \cs_if_free:NF #1
1750       {
1751         \__msg_kernel_error:nnxx { kernel } { command-already-defined }
1752           { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1753       }
1754   }
1755 ⟨*package⟩
1756 \tex_ifodd:D \l@expl@log@functions@bool
1757   \cs_set_protected:Npn \__chk_if_free_cs:N #1
1758     {
1759       \cs_if_free:NF #1
1760         {
1761           \__msg_kernel_error:nnxx { kernel } { command-already-defined }
1762             { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1763         }
```

```
1764          \__chk_log:x { Defining~\token_to_str:N #1~ \msg_line_context: }
1765        }
1766    \fi:
1767  ⟨/package⟩
1768  \cs_set_protected:Npn \__chk_if_free_cs:c
1769    { \exp_args:Nc \__chk_if_free_cs:N }
```

(*End definition for* __chk_if_free_cs:N.)

__chk_if_exist_var:N   Create the checking function for variable definitions when the option is set.

```
1770  ⟨*package⟩
1771  \tex_ifodd:D \l@expl@check@declarations@bool
1772    \cs_set_protected:Npn \__chk_if_exist_var:N #1
1773      {
1774        \cs_if_exist:NF #1
1775          {
1776            \__msg_kernel_error:nnx { check } { non-declared-variable }
1777              { \token_to_str:N #1 }
1778          }
1779      }
1780  \fi:
1781  ⟨/package⟩
```

(*End definition for* __chk_if_exist_var:N.)

__chk_if_exist_cs:N   This function issues an error message when the control sequence in its argument does
__chk_if_exist_cs:c   not exist.

```
1782  \cs_set_protected:Npn \__chk_if_exist_cs:N #1
1783    {
1784      \cs_if_exist:NF #1
1785        {
1786          \__msg_kernel_error:nnx { kernel } { command-not-defined }
1787            { \token_to_str:N #1 }
1788        }
1789    }
1790  \cs_set_protected:Npn \__chk_if_exist_cs:c
1791    { \exp_args:Nc \__chk_if_exist_cs:N }
```

(*End definition for* __chk_if_exist_cs:N.)

3.10 More new definitions

\cs_new_nopar:Npn   Function which check that the control sequence is free before defining it.
\cs_new_nopar:Npx
\cs_new:Npn
\cs_new:Npx
\cs_new_protected_nopar:Npn
\cs_new_protected_nopar:Npx
\cs_new_protected:Npn
\cs_new_protected:Npx
__cs_tmp:w

```
1792  \cs_set:Npn \__cs_tmp:w #1#2
1793    {
1794      \cs_set_protected:Npn #1 ##1
1795        {
1796          \__chk_if_free_cs:N ##1
1797          #2 ##1
1798        }
1799    }
1800  \__cs_tmp:w \cs_new_nopar:Npn              \cs_gset_nopar:Npn
1801  \__cs_tmp:w \cs_new_nopar:Npx              \cs_gset_nopar:Npx
1802  \__cs_tmp:w \cs_new:Npn                    \cs_gset:Npn
```

```
1803 \__cs_tmp:w \cs_new:Npx                          \cs_gset:Npx
1804 \__cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
1805 \__cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
1806 \__cs_tmp:w \cs_new_protected:Npn        \cs_gset_protected:Npn
1807 \__cs_tmp:w \cs_new_protected:Npx        \cs_gset_protected:Npx
```

(*End definition for* \cs_new_nopar:Npn *and others. These functions are documented on page 11.*)

\cs_set_nopar:cpn
\cs_set_nopar:cpx
\cs_gset_nopar:cpn
\cs_gset_nopar:cpx
\cs_new_nopar:cpn
\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*⟩ will turn ⟨*string*⟩ into a csname and then assign ⟨*rep-text*⟩ to it by using \cs_set_nopar:Npn. This means that there might be a parameter string between the two arguments.

```
1808 \cs_set:Npn \__cs_tmp:w #1#2
1809   { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
1810 \__cs_tmp:w \cs_set_nopar:cpn  \cs_set_nopar:Npn
1811 \__cs_tmp:w \cs_set_nopar:cpx  \cs_set_nopar:Npx
1812 \__cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
1813 \__cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
1814 \__cs_tmp:w \cs_new_nopar:cpn  \cs_new_nopar:Npn
1815 \__cs_tmp:w \cs_new_nopar:cpx  \cs_new_nopar:Npx
```

(*End definition for* \cs_set_nopar:cpn *and others. These functions are documented on page 11.*)

\cs_set:cpn
\cs_set:cpx
\cs_gset:cpn
\cs_gset:cpx
\cs_new:cpn
\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.

```
1816 \__cs_tmp:w \cs_set:cpn  \cs_set:Npn
1817 \__cs_tmp:w \cs_set:cpx  \cs_set:Npx
1818 \__cs_tmp:w \cs_gset:cpn \cs_gset:Npn
1819 \__cs_tmp:w \cs_gset:cpx \cs_gset:Npx
1820 \__cs_tmp:w \cs_new:cpn  \cs_new:Npn
1821 \__cs_tmp:w \cs_new:cpx  \cs_new:Npx
```

(*End definition for* \cs_set:cpn *and others. These functions are documented on page 11.*)

\cs_set_protected_nopar:cpn
\cs_set_protected_nopar:cpx
\cs_gset_protected_nopar:cpn
\cs_gset_protected_nopar:cpx
\cs_new_protected_nopar:cpn
\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.

```
1822 \__cs_tmp:w \cs_set_protected_nopar:cpn  \cs_set_protected_nopar:Npn
1823 \__cs_tmp:w \cs_set_protected_nopar:cpx  \cs_set_protected_nopar:Npx
1824 \__cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
1825 \__cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
1826 \__cs_tmp:w \cs_new_protected_nopar:cpn  \cs_new_protected_nopar:Npn
1827 \__cs_tmp:w \cs_new_protected_nopar:cpx  \cs_new_protected_nopar:Npx
```

(*End definition for* \cs_set_protected_nopar:cpn *and others. These functions are documented on page 12.*)

\cs_set_protected:cpn
\cs_set_protected:cpx
\cs_gset_protected:cpn
\cs_gset_protected:cpx
\cs_new_protected:cpn
\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.

```
1828 \__cs_tmp:w \cs_set_protected:cpn  \cs_set_protected:Npn
1829 \__cs_tmp:w \cs_set_protected:cpx  \cs_set_protected:Npx
1830 \__cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
```

```
1831  \__cs_tmp:w \cs_gset_protected:cpx  \cs_gset_protected:Npx
1832  \__cs_tmp:w \cs_new_protected:cpn   \cs_new_protected:Npn
1833  \__cs_tmp:w \cs_new_protected:cpx   \cs_new_protected:Npx
```

(*End definition for* `\cs_set_protected:cpn` *and others. These functions are documented on page 11.*)

3.11 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".

```
1834  \cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }
1835  \cs_new_protected:Npn \cs_set_eq:cN { \exp_args:Nc  \cs_set_eq:NN }
1836  \cs_new_protected:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
1837  \cs_new_protected:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
1838  \cs_new_protected:Npn \cs_gset_eq:NN { \tex_global:D  \cs_set_eq:NN }
1839  \cs_new_protected:Npn \cs_gset_eq:Nc { \exp_args:NNc  \cs_gset_eq:NN }
1840  \cs_new_protected:Npn \cs_gset_eq:cN { \exp_args:Nc   \cs_gset_eq:NN }
1841  \cs_new_protected:Npn \cs_gset_eq:cc { \exp_args:Ncc  \cs_gset_eq:NN }
1842  \cs_new_protected:Npn \cs_new_eq:NN #1
1843    {
1844      \__chk_if_free_cs:N #1
1845      \tex_global:D \cs_set_eq:NN #1
1846    }
1847  \cs_new_protected:Npn \cs_new_eq:cN { \exp_args:Nc  \cs_new_eq:NN }
1848  \cs_new_protected:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
1849  \cs_new_protected:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }
```

(*End definition for* `\cs_set_eq:NN`, `\cs_gset_eq:NN`, *and* `\cs_new_eq:NN`. *These functions are documented on page 15.*)

3.12 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.

```
1850  \cs_new_protected:Npn \cs_undefine:N #1
1851    { \cs_gset_eq:NN #1 \tex_undefined:D }
1852  \cs_new_protected:Npn \cs_undefine:c #1
1853    {
1854      \if_cs_exist:w #1 \cs_end:
1855        \exp_after:wN \use:n
1856      \else:
1857        \exp_after:wN \use_none:n
1858      \fi:
1859      { \cs_gset_eq:cN {#1} \tex_undefined:D }
1860    }
```

(*End definition for* `\cs_undefine:N`. *This function is documented on page 15.*)

3.13 Generating parameter text from argument count

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.

```
1861 \cs_set_protected:Npn \__cs_parm_from_arg_count:nnF #1#2
1862   {
1863     \exp_args:Nx \__cs_parm_from_arg_count_test:nnF
1864       {
1865         \exp_after:wN \exp_not:n
1866         \if_case:w \__int_eval:w #2 \__int_eval_end:
1867           { }
1868         \or: { ##1 }
1869         \or: { ##1##2 }
1870         \or: { ##1##2##3 }
1871         \or: { ##1##2##3##4 }
1872         \or: { ##1##2##3##4##5 }
1873         \or: { ##1##2##3##4##5##6 }
1874         \or: { ##1##2##3##4##5##6##7 }
1875         \or: { ##1##2##3##4##5##6##7##8 }
1876         \or: { ##1##2##3##4##5##6##7##8##9 }
1877         \else: { \c_false_bool }
1878         \fi:
1879       }
1880     {#1}
1881   }
1882 \cs_set_protected:Npn \__cs_parm_from_arg_count_test:nnF #1#2
1883   {
1884     \if_meaning:w \c_false_bool #1
1885       \exp_after:wN \use_ii:nn
1886     \else:
1887       \exp_after:wN \use_i:nn
1888     \fi:
1889     { #2 {#1} }
1890   }
```

(*End definition for* __cs_parm_from_arg_count:nnF *and* __cs_parm_from_arg_count_test:nnF.)

3.14 Defining functions from a given number of arguments

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.

```
1891 \cs_new:Npn \__cs_count_signature:N #1
1892   { \int_eval:n { \__cs_split_function:NN #1 \__cs_count_signature:nnN } }
1893 \cs_new:Npn \__cs_count_signature:nnN #1#2#3
1894   {
```

```
1895        \if_meaning:w \c_true_bool #3
1896          \tl_count:n {#2}
1897        \else:
1898          \c_minus_one
1899        \fi:
1900      }
1901 \cs_new:Npn \__cs_count_signature:c
1902      { \exp_args:Nc \__cs_count_signature:N }
```

(*End definition for* __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

```
1903 \cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
1904    {
1905      \__cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
1906        {
1907          \__msg_kernel_error:nnxx { kernel } { bad-number-of-arguments }
1908            { \token_to_str:N #1 } { \int_eval:n {#3} }
1909          \use_none:n
1910        }
1911      {#4}
1912    }
```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```
1913 \cs_new_protected:Npn \cs_generate_from_arg_count:cNnn
1914    { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
1915 \cs_new_protected:Npn \cs_generate_from_arg_count:Ncnn
1916    { \exp_args:NNc \cs_generate_from_arg_count:NNnn }
```

(*End definition for* \cs_generate_from_arg_count:NNnn. *This function is documented on page 14.*)

3.15 Using the signature to define functions

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:Nx
\cs_set_nopar:Nn
\cs_set_nopar:Nx
\cs_set_protected:Nn
\cs_set_protected:Nx
\cs_set_protected_nopar:Nn
\cs_set_protected_nopar:Nx
\cs_gset:Nn
\cs_gset:Nx
\cs_gset_nopar:Nn
\cs_gset_nopar:Nx
\cs_gset_protected:Nn
\cs_gset_protected:Nx
\cs_gset_protected_nopar:Nn
\cs_gset_protected_nopar:Nx
\cs_new:Nn
\cs_new:Nx
\cs_new_nopar:Nn
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
      { \__cs_count_signature:N #1 } {#2}
  }
```

271

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.

```
1917 \cs_set:Npn \__cs_tmp:w #1#2#3
1918   {
1919     \cs_new_protected:cpx { cs_ #1 : #2 }
1920       {
1921         \exp_not:N \__cs_generate_from_signature:NNn
1922         \exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
1923       }
1924   }
1925 \cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
1926   {
1927     \__cs_split_function:NN #2 \__cs_generate_from_signature:nnNNNn
1928     #1 #2
1929   }
1930 \cs_new_protected:Npn \__cs_generate_from_signature:nnNNNn #1#2#3#4#5#6
1931   {
1932     \bool_if:NTF #3
1933       {
1934         \str_if_eq_x:nnF { }
1935           { \tl_map_function:nN {#2} \__cs_generate_from_signature:n }
1936           {
1937             \__msg_kernel_error:nnx { kernel } { non-base-function }
1938               { \token_to_str:N #5 }
1939           }
1940         \cs_generate_from_arg_count:NNnn
1941           #5 #4 { \tl_count:n {#2} } {#6}
1942       }
1943       {
1944         \__msg_kernel_error:nnx { kernel } { missing-colon }
1945           { \token_to_str:N #5 }
1946       }
1947   }
1948 \cs_new:Npn \__cs_generate_from_signature:n #1
1949   {
1950     \if:w n #1 \else: \if:w N #1 \else:
1951     \if:w T #1 \else: \if:w F #1 \else: #1 \fi: \fi: \fi: \fi:
1952   }
```

Then we define the 24 variants beginning with `N`.

```
1953 \__cs_tmp:w { set }                  { Nn } { Npn }
1954 \__cs_tmp:w { set }                  { Nx } { Npx }
1955 \__cs_tmp:w { set_nopar }            { Nn } { Npn }
1956 \__cs_tmp:w { set_nopar }            { Nx } { Npx }
1957 \__cs_tmp:w { set_protected }        { Nn } { Npn }
1958 \__cs_tmp:w { set_protected }        { Nx } { Npx }
1959 \__cs_tmp:w { set_protected_nopar }  { Nn } { Npn }
1960 \__cs_tmp:w { set_protected_nopar }  { Nx } { Npx }
1961 \__cs_tmp:w { gset }                 { Nn } { Npn }
1962 \__cs_tmp:w { gset }                 { Nx } { Npx }
1963 \__cs_tmp:w { gset_nopar }           { Nn } { Npn }
1964 \__cs_tmp:w { gset_nopar }           { Nx } { Npx }
1965 \__cs_tmp:w { gset_protected }       { Nn } { Npn }
```

```
1966 \__cs_tmp:w { gset_protected }       { Nx } { Npx }
1967 \__cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
1968 \__cs_tmp:w { gset_protected_nopar } { Nx } { Npx }
1969 \__cs_tmp:w { new }                   { Nn } { Npn }
1970 \__cs_tmp:w { new }                   { Nx } { Npx }
1971 \__cs_tmp:w { new_nopar }             { Nn } { Npn }
1972 \__cs_tmp:w { new_nopar }             { Nx } { Npx }
1973 \__cs_tmp:w { new_protected }         { Nn } { Npn }
1974 \__cs_tmp:w { new_protected }         { Nx } { Npx }
1975 \__cs_tmp:w { new_protected_nopar }   { Nn } { Npn }
1976 \__cs_tmp:w { new_protected_nopar }   { Nx } { Npx }
```

(*End definition for* `\cs_set:Nn` *and others. These functions are documented on page 13.*)

The 24 c variants simply use `\exp_args:Nc`.

<table>
<tr><td>\cs_set:cn</td></tr>
<tr><td>\cs_set:cx</td></tr>
<tr><td>\cs_set_nopar:cn</td></tr>
<tr><td>\cs_set_nopar:cx</td></tr>
<tr><td>\cs_set_protected:cn</td></tr>
<tr><td>\cs_set_protected:cx</td></tr>
<tr><td>\cs_set_protected_nopar:cn</td></tr>
<tr><td>\cs_set_protected_nopar:cx</td></tr>
<tr><td>\cs_gset:cn</td></tr>
<tr><td>\cs_gset:cx</td></tr>
<tr><td>\cs_gset_nopar:cn</td></tr>
<tr><td>\cs_gset_nopar:cx</td></tr>
<tr><td>\cs_gset_protected:cn</td></tr>
<tr><td>\cs_gset_protected:cx</td></tr>
<tr><td>\cs_gset_protected_nopar:cn</td></tr>
<tr><td>\cs_gset_protected_nopar:cx</td></tr>
<tr><td>\cs_new:cn</td></tr>
<tr><td>\cs_new:cx</td></tr>
<tr><td>\cs_new_nopar:cn</td></tr>
<tr><td>\cs_new_nopar:cx</td></tr>
<tr><td>\cs_new_protected:cn</td></tr>
<tr><td>\cs_new_protected:cx</td></tr>
<tr><td>\cs_new_protected_nopar:cn</td></tr>
<tr><td>\cs_new_protected_nopar:cx</td></tr>
</table>

```
1977 \cs_set:Npn \__cs_tmp:w #1#2
1978   {
1979     \cs_new_protected:cpx { cs_ #1 : c #2 }
1980       {
1981         \exp_not:N \exp_args:Nc
1982         \exp_after:wN \exp_not:N \cs:w cs_ #1 : N #2 \cs_end:
1983       }
1984   }
1985 \__cs_tmp:w { set }                   { n }
1986 \__cs_tmp:w { set }                   { x }
1987 \__cs_tmp:w { set_nopar }             { n }
1988 \__cs_tmp:w { set_nopar }             { x }
1989 \__cs_tmp:w { set_protected }         { n }
1990 \__cs_tmp:w { set_protected }         { x }
1991 \__cs_tmp:w { set_protected_nopar }   { n }
1992 \__cs_tmp:w { set_protected_nopar }   { x }
1993 \__cs_tmp:w { gset }                  { n }
1994 \__cs_tmp:w { gset }                  { x }
1995 \__cs_tmp:w { gset_nopar }            { n }
1996 \__cs_tmp:w { gset_nopar }            { x }
1997 \__cs_tmp:w { gset_protected }        { n }
1998 \__cs_tmp:w { gset_protected }        { x }
1999 \__cs_tmp:w { gset_protected_nopar }  { n }
2000 \__cs_tmp:w { gset_protected_nopar }  { x }
2001 \__cs_tmp:w { new }                   { n }
2002 \__cs_tmp:w { new }                   { x }
2003 \__cs_tmp:w { new_nopar }             { n }
2004 \__cs_tmp:w { new_nopar }             { x }
2005 \__cs_tmp:w { new_protected }         { n }
2006 \__cs_tmp:w { new_protected }         { x }
2007 \__cs_tmp:w { new_protected_nopar }   { n }
2008 \__cs_tmp:w { new_protected_nopar }   { x }
```

(*End definition for* `\cs_set:cn` *and others. These functions are documented on page 13.*)

3.16 Checking control sequence equality

Check if two control sequences are identical.

<table>
<tr><td>\cs_if_eq_p:NN</td></tr>
<tr><td>\cs_if_eq_p:cN</td></tr>
<tr><td>\cs_if_eq_p:Nc</td></tr>
<tr><td>\cs_if_eq_p:cc</td></tr>
<tr><td>\cs_if_eq:NN<i>TF</i></td></tr>
<tr><td>\cs_if_eq:cN<i>TF</i></td></tr>
<tr><td>\cs_if_eq:Nc<i>TF</i></td></tr>
<tr><td>\cs_if_eq:cc<i>TF</i></td></tr>
</table>

```
2009 \prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
```

```
2010    {
2011        \if_meaning:w #1#2
2012            \prg_return_true: \else: \prg_return_false: \fi:
2013    }
2014 \cs_new:Npn \cs_if_eq_p:cN { \exp_args:Nc  \cs_if_eq_p:NN }
2015 \cs_new:Npn \cs_if_eq:cNTF { \exp_args:Nc  \cs_if_eq:NNTF }
2016 \cs_new:Npn \cs_if_eq:cNT  { \exp_args:Nc  \cs_if_eq:NNT }
2017 \cs_new:Npn \cs_if_eq:cNF  { \exp_args:Nc  \cs_if_eq:NNF }
2018 \cs_new:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
2019 \cs_new:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
2020 \cs_new:Npn \cs_if_eq:NcT  { \exp_args:NNc \cs_if_eq:NNT }
2021 \cs_new:Npn \cs_if_eq:NcF  { \exp_args:NNc \cs_if_eq:NNF }
2022 \cs_new:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
2023 \cs_new:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
2024 \cs_new:Npn \cs_if_eq:ccT  { \exp_args:Ncc \cs_if_eq:NNT }
2025 \cs_new:Npn \cs_if_eq:ccF  { \exp_args:Ncc \cs_if_eq:NNF }
```

(*End definition for* `\cs_if_eq:NNTF`*. This function is documented on page 20.*)

3.17   Diagnostic functions

`\__kernel_register_show:N`
`\__kernel_register_show:c`
Simply using the `\showthe` primitive does not allow for line-wrapping, so instead use `\__-msg_show_variable:NNNnn` (defined in l3msg). This checks that the variable exists (using `\cs_if_exist:NTF`), then displays the third argument, namely `>~`⟨*variable*⟩`=`⟨*value*⟩.

```
2026 \cs_new_protected:Npn \__kernel_register_show:N #1
2027    {
2028        \__msg_show_variable:NNNnn #1 \cs_if_exist:NTF ? { }
2029            { > ~ \token_to_str:N #1 = \tex_the:D #1 }
2030    }
2031 \cs_new_protected:Npn \__kernel_register_show:c
2032    { \exp_args:Nc \__kernel_register_show:N }
```

(*End definition for* `\__kernel_register_show:N`*.*)

`\cs_show:N`
`\cs_show:c`
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 `\iow_wrap:nnnN` for line-wrapping. The `\cs_show:c` command converts its argument to a control sequence within a group to avoid showing `\relax` for undefined control sequences.

```
2033 \cs_new_protected:Npn \cs_show:N #1
2034    { \__msg_show_wrap:n { > ~ \token_to_str:N #1 = \cs_meaning:N #1 } }
2035 \cs_new_protected:Npn \cs_show:c
2036    { \group_begin: \exp_args:NNc \group_end: \cs_show:N }
```

(*End definition for* `\cs_show:N`*. This function is documented on page 16.*)

3.18   Doing nothing functions

`\prg_do_nothing:`
This does not fit anywhere else!

```
2037 \cs_new_nopar:Npn \prg_do_nothing: { }
```

(*End definition for* `\prg_do_nothing:`*. This function is documented on page 9.*)

3.19 Breaking out of mapping functions

__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.

```
2038 \cs_new_eq:NN \__prg_break_point:Nn \use_ii:nn
2039 \cs_new:Npn \__prg_map_break:Nn #1#2#3 \__prg_break_point:Nn #4#5
2040   {
2041     #5
2042     \if_meaning:w #1 #4
2043       \exp_after:wN \use_iii:nnn
2044     \fi:
2045     \__prg_map_break:Nn #1 {#2}
2046   }
```

(*End definition for* `\__prg_break_point:Nn` *and* `\__prg_map_break:Nn`.)

__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.

```
2047 \cs_new_eq:NN \__prg_break_point: \prg_do_nothing:
2048 \cs_new:Npn \__prg_break: #1 \__prg_break_point: { }
2049 \cs_new:Npn \__prg_break:n #1#2 \__prg_break_point: {#1}
```

(*End definition for* `\__prg_break_point:` , `\__prg_break:` , *and* `\__prg_break:n`.)

```
2050 ⟨/initex | package⟩
```

4 **l3expan** implementation

```
2051 ⟨*initex | package⟩
```

```
2052 ⟨@@=exp⟩
```

\exp_after:wN
\exp_not:N
\exp_not:n
These are defined in l3basics.

(*End definition for* `\exp_after:wN`, `\exp_not:N`, *and* `\exp_not:n`. *These functions are documented on page 30.*)

4.1 General expansion

In this section a general mechanism for defining functions to handle argument handling 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 4.3. In section 4.2 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, as it is needed "early". This is just a reminder that is the case!

(*End definition for* `\l__exp_internal_tl`.)

This code uses internal functions with names that start with `\::` to perform the expansions. All macros are `long` as this turned out to be desirable 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 will require 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.

```
2053 \cs_new:Npn \__exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }
2054 \cs_new:Npn \__exp_arg_next:Nnn #1#2#3 { #2 \::: { #3 #1 } }
```

(*End definition for* `\__exp_arg_next:nnn` *and* `\__exp_arg_next:Nnn`.)

`\:::` The end marker is just another name for the identity function.

```
2055 \cs_new:Npn \::: #1 {#1}
```

(*End definition for* `\:::`.)

`\::n` This function is used to skip an argument that doesn't need to be expanded.

```
2056 \cs_new:Npn \::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(*End definition for* `\::n`.)

`\::N` This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```
2057 \cs_new:Npn \::N #1 \::: #2#3 { #1 \::: {#2#3} }
```

(*End definition for* `\::N`.)

`\::p` This function is used to skip an argument that is delimited by a left brace and doesn't need to be expanded. It should not be wrapped in braces in the result.

```
2058 \cs_new:Npn \::p #1 \::: #2#3# { #1 \::: {#2#3} }
```

(*End definition for* `\::p`.)

`\::c` This function is used to skip an argument that is turned into a control sequence without expansion.

```
2059 \cs_new:Npn \::c #1 \::: #2#3
2060   { \exp_after:wN \__exp_arg_next:Nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(*End definition for* `\::c`.)

`\::o` This function is used to expand an argument once.

```
2061 \cs_new:Npn \::o #1 \::: #2#3
2062   { \exp_after:wN \__exp_arg_next:nnn \exp_after:wN {#3} {#1} {#2} }
```

(*End definition for* \::o.)

\::f   This function is used to expand a token list until the first unexpandable token is
\exp_stop_f: found. This is achieved through \exp:w \exp_end_continue_f:w that expands ev-
erything in its way following it. This scanning procedure is terminated once the ex-
pansion hits something non-expandable or a space. We introduce \exp_stop_f: to
mark such an end of expansion marker. In the example shown earlier the scanning was
stopped once TeX had fully expanded \cs_set_eq:Nc \aaa { b \l_tmpa_tl b } into
\cs_set_eq:NN \aaa = \blurb which then turned out to contain the non-expandable
token \cs_set_eq:NN. Since the expansion of \exp:w \exp_end_continue_f:w is ⟨*null*⟩,
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 x argument type.

```
2063 \cs_new:Npn \::f #1 \::: #2#3
2064   {
2065     \exp_after:wN \__exp_arg_next:nnn
2066       \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2067       {#1} {#2}
2068   }
2069 \use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }
```

(*End definition for* \::f *and* \exp_stop_f:.)

\::x   This function is used to expand an argument fully.

```
2070 \cs_new_protected:Npn \::x #1 \::: #2#3
2071   {
2072     \cs_set_nopar:Npx \l__exp_internal_tl { {#3} }
2073     \exp_after:wN \__exp_arg_next:nnn \l__exp_internal_tl {#1} {#2}
2074   }
```

(*End definition for* \::x.)

\::v   These functions return the value of a register, i.e., one of tl, clist, int, skip, dim and
\::V   muskip. 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 will terminate using \exp_end:. The
argument is returned in braces.

```
2075 \cs_new:Npn \::V #1 \::: #2#3
2076   {
2077     \exp_after:wN \__exp_arg_next:nnn
2078       \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2079       {#1} {#2}
2080 }
2081 \cs_new:Npn \::v # 1\::: #2#3
2082   {
2083     \exp_after:wN \__exp_arg_next:nnn
2084       \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2085       {#1} {#2}
2086   }
```

(*End definition for* \::v *and* \::V.)

`\__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 do here is try to find out whether the token will expand to something else when hit with `\exp_after:wN`. The technique is to compare the meaning of the register in question when it has been prefixed with `\exp_not:N` and the register itself. If it is a macro, the prefixed `\exp_not:N` will temporarily turn it into the primitive `\scan_stop:`.

```
2087 \cs_new:Npn \__exp_eval_register:N #1
2088   {
2089     \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.

```
2090       \if_meaning:w \scan_stop: #1
2091         \__exp_eval_error_msg:w
2092       \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`.

```
2093     \else:
2094       \exp_after:wN \use_i_ii:nnn
2095     \fi:
2096     \exp_after:wN \exp_end: \tex_the:D #1
2097   }
2098 \cs_new:Npn \__exp_eval_register:c #1
2099   { \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}
```

```
2100 \cs_new:Npn \__exp_eval_error_msg:w #1 \tex_the:D #2
2101   {
2102     \fi:
2103     \fi:
2104     \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2}
2105     \exp_end:
2106   }
```

(*End definition for* `\__exp_eval_register:N` *and* `\__exp_eval_error_msg:w`.)

4.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable and therefore allow to prefix them with `\tex_global:D` for example.

`\exp_args:No`
`\exp_args:NNo`
`\exp_args:NNNo`

Those lovely runs of expansion!

```
2107 \cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
2108 \cs_new:Npn \exp_args:NNo #1#2#3
2109   { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
2110 \cs_new:Npn \exp_args:NNNo #1#2#3#4
2111   { \exp_after:wN #1 \exp_after:wN#2 \exp_after:wN #3 \exp_after:wN {#4} }
```

(*End definition for* `\exp_args:No`*,* `\exp_args:NNo`*, and* `\exp_args:NNNo`*. These functions are documented on page 27.*)

`\exp_args:Nc`
`\exp_args:cc`

In l3basics.

(*End definition for* `\exp_args:Nc` *and* `\exp_args:cc`*. These functions are documented on page 27.*)

`\exp_args:NNc`
`\exp_args:Ncc`
`\exp_args:Nccc`

Here are the functions that turn their argument into csnames but are expandable.

```
2112 \cs_new:Npn \exp_args:NNc #1#2#3
2113   { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
2114 \cs_new:Npn \exp_args:Ncc #1#2#3
2115   { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
2116 \cs_new:Npn \exp_args:Nccc #1#2#3#4
2117   {
2118     \exp_after:wN #1
2119       \cs:w #2 \exp_after:wN \cs_end:
2120       \cs:w #3 \exp_after:wN \cs_end:
2121       \cs:w #4 \cs_end:
2122   }
```

(*End definition for* `\exp_args:NNc`*,* `\exp_args:Ncc`*, and* `\exp_args:Nccc`*. These functions are documented on page 28.*)

`\exp_args:Nf`
`\exp_args:NV`
`\exp_args:Nv`

```
2123 \cs_new:Npn \exp_args:Nf #1#2
2124   { \exp_after:wN #1 \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } }
2125 \cs_new:Npn \exp_args:Nv #1#2
2126   {
2127     \exp_after:wN #1 \exp_after:wN
2128       { \exp:w \__exp_eval_register:c {#2} }
2129   }
2130 \cs_new:Npn \exp_args:NV #1#2
2131   {
2132     \exp_after:wN #1 \exp_after:wN
2133       { \exp:w \__exp_eval_register:N #2 }
2134   }
```

(*End definition for* `\exp_args:Nf`*,* `\exp_args:NV`*, and* `\exp_args:Nv`*. These functions are documented on page 27.*)

`\exp_args:NNV`
`\exp_args:NNv`
`\exp_args:NNf`
`\exp_args:NVV`
`\exp_args:Ncf`
`\exp_args:Nco`
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.

```
2135 \cs_new:Npn \exp_args:NNf #1#2#3
2136   {
2137     \exp_after:wN #1
2138     \exp_after:wN #2
2139     \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2140   }
2141 \cs_new:Npn \exp_args:NNv #1#2#3
2142   {
2143     \exp_after:wN #1
2144     \exp_after:wN #2
2145     \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2146   }
2147 \cs_new:Npn \exp_args:NNV #1#2#3
2148   {
2149     \exp_after:wN #1
2150     \exp_after:wN #2
2151     \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2152   }
2153 \cs_new:Npn \exp_args:Nco #1#2#3
2154   {
2155     \exp_after:wN #1
2156     \cs:w #2 \exp_after:wN \cs_end:
2157     \exp_after:wN {#3}
2158   }
2159 \cs_new:Npn \exp_args:Ncf #1#2#3
2160   {
2161     \exp_after:wN #1
2162     \cs:w #2 \exp_after:wN \cs_end:
2163     \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2164   }
2165 \cs_new:Npn \exp_args:NVV #1#2#3
2166   {
2167     \exp_after:wN #1
2168     \exp_after:wN { \exp:w \exp_after:wN
2169       \__exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
2170     \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2171   }
```

(*End definition for* `\exp_args:NNV` *and others. These functions are documented on page* *28.*)

A few more that we can hand-tune.

```
2172 \cs_new:Npn \exp_args:NNNV #1#2#3#4
2173   {
2174     \exp_after:wN #1
2175     \exp_after:wN #2
2176     \exp_after:wN #3
2177     \exp_after:wN { \exp:w \__exp_eval_register:N #4 }
2178   }
2179 \cs_new:Npn \exp_args:NcNc #1#2#3#4
2180   {
2181     \exp_after:wN #1
```

```
2182        \cs:w #2 \exp_after:wN \cs_end:
2183        \exp_after:wN #3
2184        \cs:w #4 \cs_end:
2185      }
2186    \cs_new:Npn \exp_args:NcNo #1#2#3#4
2187      {
2188        \exp_after:wN #1
2189        \cs:w #2 \exp_after:wN \cs_end:
2190        \exp_after:wN #3
2191        \exp_after:wN {#4}
2192      }
2193    \cs_new:Npn \exp_args:Ncco #1#2#3#4
2194      {
2195        \exp_after:wN #1
2196        \cs:w #2 \exp_after:wN \cs_end:
2197        \cs:w #3 \exp_after:wN \cs_end:
2198        \exp_after:wN {#4}
2199      }
```

(*End definition for* `\exp_args:Ncco` *and others. These functions are documented on page 28.*)

4.3 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 are all not long: they don't actually take any arguments themselves.

\exp_args:Nx

```
2200    \cs_new_protected:Npn \exp_args:Nx { \::x \::: }
```

(*End definition for* `\exp_args:Nx`*. This function is documented on page 27.*)

\exp_args:Nnc
\exp_args:Nfo
\exp_args:Nff
\exp_args:Nnf
\exp_args:Nno
\exp_args:NnV
\exp_args:Noo
\exp_args:Nof
\exp_args:Noc
\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.

```
2201    \cs_new:Npn \exp_args:Nnc { \::n \::c \::: }
2202    \cs_new:Npn \exp_args:Nfo { \::f \::o \::: }
2203    \cs_new:Npn \exp_args:Nff { \::f \::f \::: }
2204    \cs_new:Npn \exp_args:Nnf { \::n \::f \::: }
2205    \cs_new:Npn \exp_args:Nno { \::n \::o \::: }
2206    \cs_new:Npn \exp_args:NnV { \::n \::V \::: }
2207    \cs_new:Npn \exp_args:Noo { \::o \::o \::: }
2208    \cs_new:Npn \exp_args:Nof { \::o \::f \::: }
2209    \cs_new:Npn \exp_args:Noc { \::o \::c \::: }
2210    \cs_new_protected:Npn \exp_args:NNx { \::N \::x \::: }
2211    \cs_new_protected:Npn \exp_args:Ncx { \::c \::x \::: }
2212    \cs_new_protected:Npn \exp_args:Nnx { \::n \::x \::: }
2213    \cs_new_protected:Npn \exp_args:Nox { \::o \::x \::: }
2214    \cs_new_protected:Npn \exp_args:Nxo { \::x \::o \::: }
2215    \cs_new_protected:Npn \exp_args:Nxx { \::x \::x \::: }
```

(*End definition for* `\exp_args:Nnc` *and others. These functions are documented on page 28.*)

\exp_args:NNno
\exp_args:NNoo
\exp_args:Nnnc
\exp_args:Nnno
\exp_args:Nooo
\exp_args:NNNx
\exp_args:NNnx
\exp_args:NNox
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:Noox

```
2216    \cs_new:Npn \exp_args:NNno { \::N \::n \::o \::: }
2217    \cs_new:Npn \exp_args:NNoo { \::N \::o \::o \::: }
```

```
2218 \cs_new:Npn \exp_args:Nnnc { \::n \::n \::c \::: }
2219 \cs_new:Npn \exp_args:Nnno { \::n \::n \::o \::: }
2220 \cs_new:Npn \exp_args:Nooo { \::o \::o \::o \::: }
2221 \cs_new_protected:Npn \exp_args:NNNx { \::N \::N \::x \::: }
2222 \cs_new_protected:Npn \exp_args:NNnx { \::N \::n \::x \::: }
2223 \cs_new_protected:Npn \exp_args:NNox { \::N \::o \::x \::: }
2224 \cs_new_protected:Npn \exp_args:Nnnx { \::n \::n \::x \::: }
2225 \cs_new_protected:Npn \exp_args:Nnox { \::n \::o \::x \::: }
2226 \cs_new_protected:Npn \exp_args:Nccx { \::c \::c \::x \::: }
2227 \cs_new_protected:Npn \exp_args:Ncnx { \::c \::n \::x \::: }
2228 \cs_new_protected:Npn \exp_args:Noox { \::o \::o \::x \::: }
```

(*End definition for* \exp_args:NNno *and others. These functions are documented on page* *28.*)

4.4 Last-unbraced versions

__exp_arg_last_unbraced:nn
\::f_unbraced
\::o_unbraced
\::V_unbraced
\::v_unbraced
\::x_unbraced

There are a few places where the last argument needs to be available unbraced. First some helper macros.

```
2229 \cs_new:Npn \__exp_arg_last_unbraced:nn #1#2 { #2#1 }
2230 \cs_new:Npn \::f_unbraced \::: #1#2
2231   {
2232     \exp_after:wN \__exp_arg_last_unbraced:nn
2233       \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } {#1}
2234   }
2235 \cs_new:Npn \::o_unbraced \::: #1#2
2236   { \exp_after:wN \__exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
2237 \cs_new:Npn \::V_unbraced \::: #1#2
2238   {
2239     \exp_after:wN \__exp_arg_last_unbraced:nn
2240       \exp_after:wN { \exp:w \__exp_eval_register:N #2 } {#1}
2241   }
2242 \cs_new:Npn \::v_unbraced \::: #1#2
2243   {
2244     \exp_after:wN \__exp_arg_last_unbraced:nn
2245       \exp_after:wN { \exp:w \__exp_eval_register:c {#2} } {#1}
2246   }
2247 \cs_new_protected:Npn \::x_unbraced \::: #1#2
2248   {
2249     \cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n {#1} #2 }
2250     \l__exp_internal_tl
2251   }
```

(*End definition for* __exp_arg_last_unbraced:nn *and others.*)

\exp_last_unbraced:NV
\exp_last_unbraced:Nv
\exp_last_unbraced:Nf
\exp_last_unbraced:No
\exp_last_unbraced:Nco
\exp_last_unbraced:NcV
\exp_last_unbraced:NNV
\exp_last_unbraced:NNo
\exp_last_unbraced:NNNV
\exp_last_unbraced:NNNo
\exp_last_unbraced:Nno
\exp_last_unbraced:Noo
\exp_last_unbraced:Nfo
\exp_last_unbraced:NnNo
\exp_last_unbraced:Nx

Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```
2252 \cs_new:Npn \exp_last_unbraced:NV #1#2
2253   { \exp_after:wN #1 \exp:w \__exp_eval_register:N #2 }
2254 \cs_new:Npn \exp_last_unbraced:Nv #1#2
2255   { \exp_after:wN #1 \exp:w \__exp_eval_register:c {#2} }
2256 \cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
2257 \cs_new:Npn \exp_last_unbraced:Nf #1#2
2258   { \exp_after:wN #1 \exp:w \exp_end_continue_f:w #2 }
2259 \cs_new:Npn \exp_last_unbraced:Nco #1#2#3
```

```
2260     { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
2261 \cs_new:Npn \exp_last_unbraced:NcV #1#2#3
2262   {
2263     \exp_after:wN #1
2264     \cs:w #2 \exp_after:wN \cs_end:
2265     \exp:w \__exp_eval_register:N #3
2266   }
2267 \cs_new:Npn \exp_last_unbraced:NNV #1#2#3
2268   {
2269     \exp_after:wN #1
2270     \exp_after:wN #2
2271     \exp:w \__exp_eval_register:N #3
2272   }
2273 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3
2274   { \exp_after:wN #1 \exp_after:wN #2 #3 }
2275 \cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
2276   {
2277     \exp_after:wN #1
2278     \exp_after:wN #2
2279     \exp_after:wN #3
2280     \exp:w \__exp_eval_register:N #4
2281   }
2282 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
2283   { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
2284 \cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \::: }
2285 \cs_new:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \::: }
2286 \cs_new:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \::: }
2287 \cs_new:Npn \exp_last_unbraced:NnNo { \::n \::N \::o_unbraced \::: }
2288 \cs_new_protected:Npn \exp_last_unbraced:Nx { \::x_unbraced \::: }
```

(*End definition for* `\exp_last_unbraced:NV` *and others. These functions are documented on page [29](#).*)

`\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.

```
2289 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
2290   { \exp_after:wN \__exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
2291 \cs_new:Npn \__exp_last_two_unbraced:noN #1#2#3
2292     { \exp_after:wN #3 #2 #1 }
```

(*End definition for* `\exp_last_two_unbraced:Noo` *and* `\__exp_last_two_unbraced:noN`. *These functions are documented on page [29](#).*)

4.5 Preventing expansion

`\exp_not:o`
`\exp_not:c`
`\exp_not:f`
`\exp_not:V`
`\exp_not:v`

```
2293 \cs_new:Npn \exp_not:o #1 { \etex_unexpanded:D \exp_after:wN {#1} }
2294 \cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
2295 \cs_new:Npn \exp_not:f #1
2296   { \etex_unexpanded:D \exp_after:wN { \exp:w \exp_end_continue_f:w #1 } }
2297 \cs_new:Npn \exp_not:V #1
```

```
2298    {
2299      \etex_unexpanded:D \exp_after:wN
2300        { \exp:w \__exp_eval_register:N #1 }
2301    }
2302 \cs_new:Npn \exp_not:v #1
2303    {
2304      \etex_unexpanded:D \exp_after:wN
2305        { \exp:w \__exp_eval_register:c {#1} }
2306    }
```

(*End definition for* `\exp_not:o` *and others. These functions are documented on page 31.*)

4.6 Controlled expansion

`\exp:w`
`\exp_end:`
`\exp_end_continue_f:w`
`\exp_end_continue_f:nw`
To trigger a sequence of "arbitrary" 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) that 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.

```
2307 %\cs_new_eq:NN \exp:w    \tex_romannumeral:D
```

So to stop the expansion sequence in a controlled way all we need to provide is `\c_-`
`zero` as part of expanded tokens. As this is an integer constant it will immediately stop
`\tex_romannumerl:D`'s search for a number.

```
2308 %\cs_new_eq:NN \exp_end: \c_zero
```

(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*⟩ will continue searching for an optional space (and it will continue expansion doing that) — see TeXbook page 269 for details.

```
2309 \tex_catcode:D '\^^@=13
2310 \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.[6]

```
2311 \cs_new:Npn ^^@{\expansionERROR}
```

```
2312 \cs_new:Npn \exp_end_continue_f:nw #1 { '^^@ #1 }
2313 \tex_catcode:D '\^^@=15
```

(*End definition for* `\exp:w` *and others. These functions are documented on page 31.*)

---

[6]Need to get a real error message.

4.7 Defining function variants

\cs_generate_variant:Nn #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:Npx or \cs_new_protected:Npx, 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.

```
2315 \cs_new_protected:Npn \cs_generate_variant:Nn #1#2
2316   {
2317     \__chk_if_exist_cs:N #1
2318     \__cs_generate_variant:N #1
2319     \exp_after:wN \__cs_split_function:NN
2320     \exp_after:wN #1
2321     \exp_after:wN \__cs_generate_variant:nnNN
2322     \exp_after:wN #1
2323     \tl_to_str:n {#2} , \scan_stop: , \q_recursion_stop
2324   }
```

(*End definition for* \cs_generate_variant:Nn*. This function is documented on page 25.*)

__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 all primitive TeX conditionals are expandable.

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 fist 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 \q_mark is taken as an argument of the wwNw auxiliary, and #3 is \cs_new_protected:Npx, otherwise it is \cs_new:Npx.

```
2325 \cs_new_protected:Npx \__cs_generate_variant:N #1
2326   {
2327     \exp_not:N \exp_after:wN \exp_not:N \if_meaning:w
2328       \exp_not:N \exp_not:N #1 #1
2329       \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx
2330     \exp_not:N \else:
2331     \exp_not:N \exp_after:wN \exp_not:N \__cs_generate_variant:ww
2332       \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma }
2333         \exp_not:N \q_mark
2334       \exp_not:N \q_mark \cs_new_protected:Npx
2335       \tl_to_str:n { pr }
2336       \exp_not:N \q_mark \cs_new:Npx
2337         \exp_not:N \q_stop
2338     \exp_not:N \fi:
```

```
2339     }
2340   \use:x
2341     {
2342       \cs_new_protected:Npn \exp_not:N \__cs_generate_variant:ww
2343         ##1 \tl_to_str:n { ma } ##2 \exp_not:N \q_mark
2344     }
2345     { \__cs_generate_variant:wwNw #1 }
2346   \use:x
2347     {
2348       \cs_new_protected:Npn \exp_not:N \__cs_generate_variant:wwNw
2349         ##1 \tl_to_str:n { pr } ##2 \exp_not:N \q_mark
2350         ##3 ##4 \exp_not:N \q_stop
2351     }
2352     { \cs_set_eq:NN \__cs_tmp:w #3 }
```

(*End 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.

```
2353  \cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
2354    {
2355      \if_meaning:w \c_false_bool #3
2356        \__msg_kernel_error:nnx { kernel } { missing-colon }
2357          { \token_to_str:c {#1} }
2358        \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2359      \fi:
2360      \__cs_generate_variant:Nnnw #4 {#1}{#2}
2361    }
```

(*End 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}, it would be better to 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.

286

- In `\cs_generate_variant:Nn \foo:on {ox}`, the function `\foo:ox` should be defined using `\exp_args:Nnx`, not `\exp_args:Nox`, to avoid double o expansion.

- Lastly, `\cs_generate_variant:Nn \foo:on {xn}` should trigger an error, because we do not have a means to replace o-expansion by x-expansion.

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 two cases: `N` and `p`-type arguments). A common trailing part is ignored.

We compare the base and variant signatures one character at a time within x-expansion. The result is given to `\__cs_generate_variant:wwNN` in the form ⟨*processed variant signature*⟩ `\q_mark` ⟨*errors*⟩ `\q_stop` ⟨*base function*⟩ ⟨*new function*⟩. If all went well, ⟨*errors*⟩ is empty; otherwise, it is a kernel error message, followed by some clean-up code (`\use_none:nnnn`).

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`.

```
2362 \cs_new_protected:Npn \__cs_generate_variant:Nnnw #1#2#3#4 ,
2363   {
2364     \if_meaning:w \scan_stop: #4
2365       \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2366     \fi:
2367     \use:x
2368       {
2369         \exp_not:N \__cs_generate_variant:wwNN
2370         \__cs_generate_variant_loop:nNwN { }
2371           #4
2372           \__cs_generate_variant_loop_end:nwwwNNnn
2373           \q_mark
2374           #3 ~
2375           { ~ { } \fi: \__cs_generate_variant_loop_long:wNNnn } ~
2376           { }
2377           \q_stop
2378         \exp_not:N #1 {#2} {#4}
2379       }
2380     \__cs_generate_variant:Nnnw #1 {#2} {#3}
2381   }
```

(*End definition for* `\__cs_generate_variant:Nnnw`.)

`\__cs_generate_variant_loop:nNwN`
`\__cs_generate_variant_loop_same:w`
`\__cs_generate_variant_loop_end:nwwwNNnn`
`\__cs_generate_variant_loop_long:wNNnn`
`\__cs_generate_variant_loop_invalid:NNwNNnn`

#1 : Last few (consecutive) letters common between the base and variant (in fact, `\__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 with a base letter of `N` or `n`. 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 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 neither n nor N. Again, 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.

```
2382 \cs_new:Npn \__cs_generate_variant_loop:nNwN #1#2#3 \q_mark #4
2383   {
2384     \if:w #2 #4
2385       \exp_after:wN \__cs_generate_variant_loop_same:w
2386     \else:
2387       \if:w N #4 \else:
2388         \if:w n #4 \else:
2389           \__cs_generate_variant_loop_invalid:NNwNNnn #4#2
2390         \fi:
2391       \fi:
2392     \fi:
2393     #1
2394     \prg_do_nothing:
2395     #2
2396     \__cs_generate_variant_loop:nNwN { } #3 \q_mark
2397   }
2398 \cs_new:Npn \__cs_generate_variant_loop_same:w
2399     #1 \prg_do_nothing: #2#3#4
2400   {
2401     #3 { #1 \__cs_generate_variant_same:N #2 }
2402   }
2403 \cs_new:Npn \__cs_generate_variant_loop_end:nwwwNNnn
2404     #1#2 \q_mark #3 ~ #4 \q_stop #5#6#7#8
2405   {
2406     \scan_stop: \scan_stop: \fi:
2407     \exp_not:N \q_mark
2408     \exp_not:N \q_stop
```

```
2409        \exp_not:N #6
2410        \exp_not:c { #7 : #8 #1 #3 }
2411      }
2412  \cs_new:Npn \__cs_generate_variant_loop_long:wNNnn #1 \q_stop #2#3#4#5
2413    {
2414      \exp_not:n
2415        {
2416          \q_mark
2417          \__msg_kernel_error:nnxx { kernel } { variant-too-long }
2418            {#5} { \token_to_str:N #3 }
2419          \use_none:nnnn
2420          \q_stop
2421          #3
2422          #3
2423        }
2424    }
2425  \cs_new:Npn \__cs_generate_variant_loop_invalid:NNwNNnn
2426      #1#2 \fi: \fi: \fi: #3 \q_stop #4#5#6#7
2427    {
2428      \fi: \fi: \fi:
2429      \exp_not:n
2430        {
2431          \q_mark
2432          \__msg_kernel_error:nnxxxx { kernel } { invalid-variant }
2433            {#7} { \token_to_str:N #5 } {#1} {#2}
2434          \use_none:nnnn
2435          \q_stop
2436          #5
2437          #5
2438        }
2439    }
```

(*End 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.

```
2440  \cs_new:Npn \__cs_generate_variant_same:N #1
2441    {
2442      \if:w N #1
2443        N
2444      \else:
2445        \if:w p #1
2446          p
2447        \else:
2448          n
2449        \fi:
2450      \fi:
2451    }
```

(*End definition for* __cs_generate_variant_same:N.)

__cs_generate_variant:wwNN If the variant form has already been defined, log its existence. 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:Npx`. Then define the variant by combining the `\exp_args:N #3` variant and the base function.

```
2452 \cs_new_protected:Npn \__cs_generate_variant:wwNN
2453     #1 \q_mark #2 \q_stop #3#4
2454   {
2455     #2
2456     \cs_if_free:NTF #4
2457       {
2458         \group_begin:
2459           \__cs_generate_internal_variant:n {#1}
2460           \__cs_tmp:w #4 { \exp_not:c { exp_args:N #1 } \exp_not:N #3 }
2461         \group_end:
2462       }
2463       {
2464         \__chk_log:x
2465           {
2466             Variant~\token_to_str:N #4~%
2467             already~defined;~ not~ changing~ it~ \msg_line_context:
2468           }
2469       }
2470   }
```

(*End definition for* `\__cs_generate_variant:wwNN`.)

`\__cs_generate_internal_variant:n`
`\__cs_generate_internal_variant:wwnw`
`\__cs_generate_internal_variant_loop:n`

Test if `\exp_args:N #1` is already defined and if not define it via the `\::` commands using the chars in `#1`. If `#1` contains an `x` (this is the place where having converted the original comma-list argument to a string is very important), the result should be protected, and the next variant to be defined using that internal variant should be protected.

```
2471 \cs_new_protected:Npx \__cs_generate_internal_variant:n #1
2472   {
2473     \exp_not:N \__cs_generate_internal_variant:wwnNwnn
2474       #1 \exp_not:N \q_mark
2475         { \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx }
2476         \cs_new_protected:cpx
2477       \token_to_str:N x \exp_not:N \q_mark
2478         { }
2479         \cs_new:cpx
2480     \exp_not:N \q_stop
2481       { exp_args:N #1 }
2482       {
2483         \exp_not:N \__cs_generate_internal_variant_loop:n #1
2484           { : \exp_not:N \use_i:nn }
2485       }
2486   }
2487 \use:x
2488   {
2489     \cs_new_protected:Npn \exp_not:N \__cs_generate_internal_variant:wwnNwnn
2490         ##1 \token_to_str:N x ##2 \exp_not:N \q_mark
2491         ##3 ##4 ##5 \exp_not:N \q_stop ##6 ##7
2492   }
2493   {
2494     #3
2495     \cs_if_free:cT {#6} { #4 {#6} {#7} }
2496   }
```

290

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.

```
2497 \cs_new:Npn \__cs_generate_internal_variant_loop:n #1
2498   {
2499     \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
2500     \__cs_generate_internal_variant_loop:n
2501   }
```

(*End definition for* __cs_generate_internal_variant:n, __cs_generate_internal_variant:wwnw, *and* __cs_generate_internal_variant_loop:n.)

```
2502 ⟨/initex | package⟩
```

5   l3prg implementation

The following test files are used for this code: m3prg001.lvt,m3prg002.lvt,m3prg003.lvt.

```
2503 ⟨*initex | package⟩
```

5.1   Primitive conditionals

\if_bool:N   Those two primitive TEX conditionals are synonyms.
\if_predicate:w
```
2504 \cs_new_eq:NN \if_bool:N      \tex_ifodd:D
2505 \cs_new_eq:NN \if_predicate:w \tex_ifodd:D
```

(*End definition for* \if_bool:N *and* \if_predicate:w. *These functions are documented on page 40.*)

5.2   Defining a set of conditional functions

\prg_set_conditional:Npnn   These are all defined in l3basics, as they are needed "early". This is just a reminder!
\prg_new_conditional:Npnn
\prg_set_protected_conditional:Npnn   (*End definition for* \prg_set_conditional:Npnn *and others. These functions are documented on page*
\prg_new_protected_conditional:Npnn   *34.*)
\prg_set_conditional:Nnn
\prg_new_conditional:Nnn
\prg_set_protected_conditional:Nnn
\prg_new_protected_conditional:Nnn
\prg_set_eq_conditional:NNn \bool_new:N
\prg_new_eq_conditional:NNn \bool_new:c
\prg_return_true:
\prg_return_false:

5.3   The boolean data type

```
2506 ⟨@@=bool⟩
```

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

```
2507 \cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
2508 \cs_generate_variant:Nn \bool_new:N { c }
```

(*End definition for* \bool_new:N. *This function is documented on page 36.*)

\bool_set_true:N   Setting is already pretty easy.
\bool_set_true:c
\bool_gset_true:N
```
2509 \cs_new_protected:Npn \bool_set_true:N #1
\bool_gset_true:c     2510   { \cs_set_eq:NN #1 \c_true_bool }
\bool_set_false:N     2511 \cs_new_protected:Npn \bool_set_false:N #1
\bool_set_false:c     2512   { \cs_set_eq:NN #1 \c_false_bool }
\bool_gset_false:N     2513 \cs_new_protected:Npn \bool_gset_true:N #1
\bool_gset_false:c     2514   { \cs_gset_eq:NN #1 \c_true_bool }
```

```
2515  \cs_new_protected:Npn \bool_gset_false:N #1
2516    { \cs_gset_eq:NN #1 \c_false_bool }
2517  \cs_generate_variant:Nn \bool_set_true:N   { c }
2518  \cs_generate_variant:Nn \bool_set_false:N  { c }
2519  \cs_generate_variant:Nn \bool_gset_true:N  { c }
2520  \cs_generate_variant:Nn \bool_gset_false:N { c }
```

(*End definition for* \bool_set_true:N *and others. These functions are documented on page 36.*)

\bool_set_eq:NN   The usual copy code.
\bool_set_eq:cN
\bool_set_eq:Nc
```
2521  \cs_new_eq:NN \bool_set_eq:NN   \cs_set_eq:NN
```
\bool_set_eq:cc
```
2522  \cs_new_eq:NN \bool_set_eq:Nc   \cs_set_eq:Nc
```
\bool_gset_eq:NN
```
2523  \cs_new_eq:NN \bool_set_eq:cN   \cs_set_eq:cN
```
\bool_gset_eq:cN
```
2524  \cs_new_eq:NN \bool_set_eq:cc   \cs_set_eq:cc
```
\bool_gset_eq:Nc
```
2525  \cs_new_eq:NN \bool_gset_eq:NN \cs_gset_eq:NN
```
\bool_gset_eq:cc
```
2526  \cs_new_eq:NN \bool_gset_eq:Nc \cs_gset_eq:Nc
2527  \cs_new_eq:NN \bool_gset_eq:cN \cs_gset_eq:cN
2528  \cs_new_eq:NN \bool_gset_eq:cc \cs_gset_eq:cc
```

(*End definition for* \bool_set_eq:NN *and* \bool_gset_eq:NN. *These functions are documented on page 37.*)

\bool_set:Nn   This function evaluates a boolean expression and assigns the first argument the meaning
\bool_set:cn   \c_true_bool or \c_false_bool.
\bool_gset:Nn
```
2529  \cs_new_protected:Npn \bool_set:Nn #1#2
```
\bool_gset:cn
```
2530    { \tex_chardef:D #1 = \bool_if_p:n {#2} }
2531  \cs_new_protected:Npn \bool_gset:Nn #1#2
2532    { \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2} }
2533  \cs_generate_variant:Nn \bool_set:Nn  { c }
2534  \cs_generate_variant:Nn \bool_gset:Nn { c }
```

(*End definition for* \bool_set:Nn *and* \bool_gset:Nn. *These functions are documented on page 37.*)

Booleans are not based on token lists but do need checking: this code complements similar material in l3tl.

```
2535  ⟨*package⟩
2536  \if_bool:N \l@expl@check@declarations@bool
2537    \cs_set_protected:Npn \bool_set_true:N #1
2538      {
2539        \__chk_if_exist_var:N #1
2540        \cs_set_eq:NN #1 \c_true_bool
2541      }
2542    \cs_set_protected:Npn \bool_set_false:N #1
2543      {
2544        \__chk_if_exist_var:N #1
2545        \cs_set_eq:NN #1 \c_false_bool
2546      }
2547    \cs_set_protected:Npn \bool_gset_true:N #1
2548      {
2549        \__chk_if_exist_var:N #1
2550        \cs_gset_eq:NN #1 \c_true_bool
2551      }
2552    \cs_set_protected:Npn \bool_gset_false:N #1
2553      {
2554        \__chk_if_exist_var:N #1
```

292

```
2555        \cs_gset_eq:NN #1 \c_false_bool
2556      }
2557    \cs_set_protected:Npn \bool_set_eq:NN  #1
2558      {
2559        \__chk_if_exist_var:N #1
2560        \cs_set_eq:NN #1
2561      }
2562    \cs_set_protected:Npn \bool_gset_eq:NN  #1
2563      {
2564        \__chk_if_exist_var:N #1
2565        \cs_gset_eq:NN #1
2566      }
2567    \cs_set_protected:Npn \bool_set:Nn #1#2
2568      {
2569        \__chk_if_exist_var:N #1
2570        \tex_chardef:D #1 = \bool_if_p:n {#2}
2571      }
2572    \cs_set_protected:Npn \bool_gset:Nn #1#2
2573      {
2574        \__chk_if_exist_var:N #1
2575        \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}
2576      }
2577 \fi:
2578 ⟨/package⟩
```

\bool_if_p:N Straight forward here. We could optimize here if we wanted to as the boolean can just
\bool_if_p:c be input directly.
\bool_if:NTF
\bool_if:cTF
```
2579 \prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
2580    {
2581      \if_meaning:w \c_true_bool #1
2582        \prg_return_true:
2583      \else:
2584        \prg_return_false:
2585      \fi:
2586    }
2587 \cs_generate_variant:Nn \bool_if_p:N { c }
2588 \cs_generate_variant:Nn \bool_if:NT  { c }
2589 \cs_generate_variant:Nn \bool_if:NF  { c }
2590 \cs_generate_variant:Nn \bool_if:NTF { c }
```

(*End definition for* \bool_if:NTF. *This function is documented on page 37.*)

\bool_show:N Show the truth value of the boolean, as `true` or `false`.
\bool_show:c
\bool_show:n
__bool_to_str:n
```
2591 \cs_new_protected:Npn \bool_show:N #1
2592    {
2593      \__msg_show_variable:NNNnn #1 \bool_if_exist:NTF ? { }
2594        { > ~ \token_to_str:N #1 = \__bool_to_str:n {#1} }
2595    }
2596 \cs_new_protected:Npn \bool_show:n
2597    { \__msg_show_wrap:Nn \__bool_to_str:n }
2598 \cs_new:Npn \__bool_to_str:n #1
2599    { \bool_if:nTF {#1} { true } { false } }
2600 \cs_generate_variant:Nn \bool_show:N { c }
```

293

(*End definition for* `\bool_show:N`*,* `\bool_show:n`*, and* `\__bool_to_str:n`*. These functions are documented on page 37.*)

`\l_tmpa_bool` A few booleans just if you need them.
`\l_tmpb_bool`
`\g_tmpa_bool`   2601 `\bool_new:N \l_tmpa_bool`
`\g_tmpb_bool`   2602 `\bool_new:N \l_tmpb_bool`
                 2603 `\bool_new:N \g_tmpa_bool`
                 2604 `\bool_new:N \g_tmpb_bool`

(*End definition for* `\l_tmpa_bool` *and others. These variables are documented on page 37.*)

`\bool_if_exist_p:N` Copies of the `cs` functions defined in l3basics.
`\bool_if_exist_p:c`
`\bool_if_exist:NTF`   2605 `\prg_new_eq_conditional:NNn \bool_if_exist:N \cs_if_exist:N`
`\bool_if_exist:cTF`   2606 `  { TF , T , F , p }`
                       2607 `\prg_new_eq_conditional:NNn \bool_if_exist:c \cs_if_exist:c`
                       2608 `  { TF , T , F , p }`

(*End definition for* `\bool_if_exist:NTF`*. This function is documented on page 37.*)

5.4   Boolean expressions

`\bool_if_p:n` Evaluating the truth value of a list of predicates is done using an input syntax somewhat
`\bool_if:nTF` similar to the one found in other programming languages with (and) for grouping, !
for logical "Not", && for logical "And" and || for logical "Or". 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 GetNotNext function, which eventually
  reverses the logic compared to GetNext.

- 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 evaluating the predicates
within this sub-expression and break to the nearest Close. Then return ⟨*false*⟩.

⟨*true*⟩**Or** Current truth value is true, logical Or seen, stop evaluating the predicates
within this sub-expression and break to 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*⟩.

294

⟨*false*⟩**Close** Current truth value is false, Close seen, return ⟨*false*⟩.

We introduce an additional Stop operation with the same semantics as the Close operation.

⟨*true*⟩**Stop** Current truth value is true, return ⟨*true*⟩.

⟨*false*⟩**Stop** Current truth value is false, return ⟨*false*⟩.

The reasons for this follow below.

```
2609 \prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
2610   {
2611     \if_predicate:w \bool_if_p:n {#1}
2612       \prg_return_true:
2613     \else:
2614       \prg_return_false:
2615     \fi:
2616   }
```

(*End definition for* \bool_if:nTF. *This function is documented on page 38.*)

\bool_if_p:n First issue a \group_align_safe_begin: as we are using && as syntax shorthand for the
__bool_if_left_parentheses:wwwn And operation and we need to hide it for TeX. This will be closed at the end of the
__bool_if_right_parentheses:wwwn expression parsing (see S below).
__bool_if_or:wwwn Minimal ("short-circuit") evaluation of boolean expressions requires skipping to the
end of the current parenthesized group when ⟨*true*⟩|| is seen, but to the next || or
closing parenthesis when ⟨*false*⟩&& is seen. To avoid having separate functions for the
two cases, we transform the boolean expression by doubling each parenthesis and adding
parenthesis around each ||. This ensures that && will bind tighter than ||.

The replacement is done in three passes, for left and right parentheses and for ||.
At each pass, the part of the expression that has been transformed is stored before \q_-
nil, the rest lies until the first \q_mark, followed by an empty brace group. A trailing
marker ensures that the auxiliaries' delimited arguments will not run-away. As long as
the delimiter matches inside the expression, material is moved before \q_nil and we
continue. Afterwards, the trailing marker is taken as a delimiter, #4 is the next auxiliary,
immediately followed by a new \q_nil delimiter, which indicates that nothing has been
treated at this pass. The last step calls __bool_if_parse:NNNww which cleans up and
triggers the evaluation of the expression itself.

```
2617 \cs_new:Npn \bool_if_p:n #1
2618   {
2619     \group_align_safe_begin:
2620     \__bool_if_left_parentheses:wwwn \q_nil
2621       #1 \q_mark { }
2622       ( \q_mark { \__bool_if_right_parentheses:wwwn \q_nil }
2623       ) \q_mark { \__bool_if_or:wwwn \q_nil }
2624       || \q_mark \__bool_if_parse:NNNww
2625     \q_stop
2626   }
2627 \cs_new:Npn \__bool_if_left_parentheses:wwwn #1 \q_nil #2 ( #3 \q_mark #4
2628   { #4 \__bool_if_left_parentheses:wwwn #1 #2 (( \q_nil #3 \q_mark {#4} }
2629 \cs_new:Npn \__bool_if_right_parentheses:wwwn #1 \q_nil #2 ) #3 \q_mark #4
2630   { #4 \__bool_if_right_parentheses:wwwn #1 #2 )) \q_nil #3 \q_mark {#4} }
2631 \cs_new:Npn \__bool_if_or:wwwn #1 \q_nil #2 || #3 \q_mark #4
2632   { #4 \__bool_if_or:wwwn #1 #2 )||( \q_nil #3 \q_mark {#4} }
```

(*End definition for* `\bool_if_p:n` *and others. These functions are documented on page 38.*)

`\__bool_if_parse:NNNww` After removing extra tokens from the transformation phase, start evaluating. At the end, we will need to finish the special `align_safe` group before finally returning a `\c_true_-bool` or `\c_false_bool` as there might otherwise be something left in front in the input stream. For this we call the Stop operation, denoted simply by a `S` following the last Close operation.

```
2633 \cs_new:Npn \__bool_if_parse:NNNww #1#2#3#4 \q_mark #5 \q_stop
2634   {
2635     \__bool_get_next:NN \use_i:nn (( #4 )) S
2636   }
```

(*End definition for* `\__bool_if_parse:NNNww`.)

`\__bool_get_next:NN` The GetNext operation. This is a switch: if what follows is neither `!` nor `(`, we assume it is a predicate. The first argument is `\use_ii:nn` if the logic must eventually be reversed (after a `!`), otherwise it is `\use_i:nn`. 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.

```
2637 \cs_new:Npn \__bool_get_next:NN #1#2
2638   {
2639     \use:c
2640       {
2641         __bool_
2642         \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
2643         :Nw
2644       }
2645     #1 #2
2646   }
```

(*End definition for* `\__bool_get_next:NN`.)

`\__bool_!:Nw` The Not operation reverses the logic: discard the `!` token and call the GetNext operation with its first argument reversed.

```
2647 \cs_new:cpn { __bool_!:Nw } #1#2
2648   { \exp_after:wN \__bool_get_next:NN #1 \use_ii:nn \use_i:nn }
```

(*End definition for* `\__bool_!:Nw`.)

`\__bool_(:Nw` The Open operation starts a sub-expression after discarding the token. This is done by calling GetNext, with a post-processing step which looks for And, Or or Close afterwards.

```
2649 \cs_new:cpn { __bool_(:Nw } #1#2
2650   {
2651     \exp_after:wN \__bool_choose:NNN \exp_after:wN #1
2652     \__int_value:w \__bool_get_next:NN \use_i:nn
2653   }
```

(*End 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.

```
2654 \cs_new:cpn { __bool_p:Nw } #1
2655   { \exp_after:wN \__bool_choose:NNN \exp_after:wN #1 \__int_value:w }
```

296

(*End definition for* `\__bool_p:Nw`*.*)

`\__bool_choose:NNN` Branching the eight-way switch. The arguments are 1: `\use_i:nn` or `\use_ii:nn`, 2: 0 or 1 encoding the current truth value, 3: the next operation, And, Or, Close or Stop. If #1 is `\use_ii:nn`, the logic of #2 must be reversed.

```
2656 \cs_new:Npn \__bool_choose:NNN #1#2#3
2657   {
2658     \use:c
2659       {
2660         __bool_ #3 _
2661         #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: }
2662         :w
2663       }
2664   }
```

(*End definition for* `\__bool_choose:NNN`*.*)

`\__bool_)_0:w`
`\__bool_)_1:w`
`\__bool_S_0:w`
`\__bool_S_1:w`

Closing a group is just about returning the result. The Stop operation is similar except it closes the special alignment group before returning the boolean.

```
2665 \cs_new:cpn { __bool_)_0:w } { \c_false_bool }
2666 \cs_new:cpn { __bool_)_1:w } { \c_true_bool }
2667 \cs_new:cpn { __bool_S_0:w } { \group_align_safe_end: \c_false_bool }
2668 \cs_new:cpn { __bool_S_1:w } { \group_align_safe_end: \c_true_bool }
```

(*End definition for* `\__bool_)_0:w` *and others.*)

`\__bool_&_1:w`
`\__bool_|_0:w`

Two cases where we simply continue scanning. We must remove the second & or |.

```
2669 \cs_new:cpn { __bool_&_1:w } & { \__bool_get_next:NN \use_i:nn }
2670 \cs_new:cpn { __bool_|_0:w } | { \__bool_get_next:NN \use_i:nn }
```

(*End definition for* `\__bool_&_1:w` *and* `\__bool_|_0:w`*.*)

`\__bool_&_0:w`
`\__bool_|_1:w`
`\__bool_eval_skip_to_end_auxi:Nw`
`\__bool_eval_skip_to_end_auxii:Nw`
`\__bool_eval_skip_to_end_auxiii:Nw`

When the truth value has already been decided, we have to throw away the remainder of the current group as we are doing minimal evaluation. This is slightly tricky as there are no braces so we have to play match the () manually.

```
2671 \cs_new:cpn { __bool_&_0:w } &
2672   { \__bool_eval_skip_to_end_auxi:Nw \c_false_bool }
2673 \cs_new:cpn { __bool_|_1:w } |
2674   { \__bool_eval_skip_to_end_auxi:Nw \c_true_bool }
```

There is always at least one) waiting, namely the outer one. However, we are facing the problem that there may be more than one that need to be finished off and we have to detect the correct number of them. Here is a complicated example showing how this is done. After evaluating the following, we realize we must skip everything after the first And. Note the extra Close at the end.

```
\c_false_bool  && ((abc) && xyz) && ((xyz) && (def)))
```

First read up to the first Close. This gives us the list we first read up until the first right parenthesis so we are looking at the token list

```
((abc
```

297

This contains two Open markers so we must remove two groups. Since no evaluation of the contents is to be carried out, it doesn't matter how we remove the groups as long as we wind up with the correct result. We therefore first remove a `()` pair and what preceded the Open – but leave the contents as it may contain Open tokens itself – leaving

    (abc && xyz) && ((xyz) && (def)))

Another round of this gives us

    (abc && xyz

which still contains an Open so we remove another `()` pair, giving us

    abc && xyz && ((xyz) && (def)))

Again we read up to a Close and again find Open tokens:

    abc && xyz && ((xyz

Further reduction gives us

    (xyz && (def)))

and then

    (xyz && (def

with reduction to

    xyz && (def))

and ultimately we arrive at no Open tokens being skipped and we can finally close the group nicely.

```
2675 %% (
2676 \cs_new:Npn \__bool_eval_skip_to_end_auxi:Nw #1#2 )
2677   {
2678     \__bool_eval_skip_to_end_auxii:Nw #1#2 ( % )
2679     \q_no_value \q_stop
2680     {#2}
2681   }
```

If no right parenthesis, then `#3` is no_value and we are done, return the boolean `#1`. If there is, we need to grab a `()` pair and then recurse

```
2682 \cs_new:Npn \__bool_eval_skip_to_end_auxii:Nw #1#2 ( #3#4 \q_stop #5 % )
2683   {
2684     \quark_if_no_value:NTF #3
2685     {#1}
2686     { \__bool_eval_skip_to_end_auxiii:Nw #1 #5 }
2687   }
```

Keep the boolean, throw away anything up to the `(` as it is irrelevant, remove a `()` pair but remember to reinsert `#3` as it may contain `(` tokens!

```
2688 \cs_new:Npn \__bool_eval_skip_to_end_auxiii:Nw #1#2 ( #3 )
2689   { % (
2690     \__bool_eval_skip_to_end_auxi:Nw #1#3 )
2691   }
```

(*End definition for* `\__bool_&_0:w` *and others.*)

$\bool\_not\_p:n$ 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.

```
2692 \cs_new:Npn \bool_not_p:n #1 { \bool_if_p:n { ! ( #1 ) } }
```

(*End definition for* `\bool_not_p:n`. *This function is documented on page 38.*)

$\bool\_xor\_p:nn$ Exclusive or. If the boolean expressions have same truth value, return false, otherwise return true.

```
2693 \cs_new:Npn \bool_xor_p:nn #1#2
2694   {
2695     \int_compare:nNnTF { \bool_if_p:n {#1} } = { \bool_if_p:n {#2} }
2696       \c_false_bool
2697       \c_true_bool
2698   }
```

(*End definition for* `\bool_xor_p:nn`. *This function is documented on page 38.*)

5.5 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.

```
2699 \cs_new:Npn \bool_while_do:Nn #1#2
2700   { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
2701 \cs_new:Npn \bool_until_do:Nn #1#2
2702   { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
2703 \cs_generate_variant:Nn \bool_while_do:Nn { c }
2704 \cs_generate_variant:Nn \bool_until_do:Nn { c }
```

(*End definition for* `\bool_while_do:Nn` *and* `\bool_until_do:Nn`. *These functions are documented on page 39.*)

$\bool\_do\_while:Nn$
$\bool\_do\_while:cn$
$\bool\_do\_until:Nn$
$\bool\_do\_until:cn$
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.

```
2705 \cs_new:Npn \bool_do_while:Nn #1#2
2706   { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
2707 \cs_new:Npn \bool_do_until:Nn #1#2
2708   { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
2709 \cs_generate_variant:Nn \bool_do_while:Nn { c }
2710 \cs_generate_variant:Nn \bool_do_until:Nn { c }
```

(*End definition for* `\bool_do_while:Nn` *and* `\bool_do_until:Nn`. *These functions are documented on page 38.*)

$\bool\_while\_do:nn$
$\bool\_do\_while:nn$
$\bool\_until\_do:nn$
$\bool\_do\_until:nn$
Loop functions with the test either before or after the first body expansion.

```
2711 \cs_new:Npn \bool_while_do:nn #1#2
2712   {
2713     \bool_if:nT {#1}
2714       {
2715         #2
2716         \bool_while_do:nn {#1} {#2}
2717       }
2718   }
```

299

```
2719 \cs_new:Npn \bool_do_while:nn #1#2
2720   {
2721     #2
2722     \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
2723   }
2724 \cs_new:Npn \bool_until_do:nn #1#2
2725   {
2726     \bool_if:nF {#1}
2727       {
2728         #2
2729         \bool_until_do:nn {#1} {#2}
2730       }
2731   }
2732 \cs_new:Npn \bool_do_until:nn #1#2
2733   {
2734     #2
2735     \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2}  }
2736   }
```

(*End definition for* \bool_while_do:nn *and others. These functions are documented on page* *39.*)

5.6   Producing multiple copies

```
2737 ⟨@@=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 will severely affect 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.

```
2738 \cs_new:Npn \prg_replicate:nn #1
2739   {
2740     \exp:w
2741       \exp_after:wN \__prg_replicate_first:N
2742         \__int_value:w \__int_eval:w #1 \__int_eval_end:
2743       \cs_end:
2744   }
2745 \cs_new:Npn \__prg_replicate:N #1
```

```
2746      { \cs:w __prg_replicate_#1 :n \__prg_replicate:N }
2747  \cs_new:Npn \__prg_replicate_first:N #1
2748      { \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.

```
2749  \cs_new:Npn \__prg_replicate_ :n #1 { \cs_end: }
2750  \cs_new:cpn { __prg_replicate_0:n } #1
2751      { \cs_end: {#1#1#1#1#1#1#1#1#1#1} }
2752  \cs_new:cpn { __prg_replicate_1:n } #1
2753      { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1 }
2754  \cs_new:cpn { __prg_replicate_2:n } #1
2755      { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1 }
2756  \cs_new:cpn { __prg_replicate_3:n } #1
2757      { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1 }
2758  \cs_new:cpn { __prg_replicate_4:n } #1
2759      { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1 }
2760  \cs_new:cpn { __prg_replicate_5:n } #1
2761      { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1 }
2762  \cs_new:cpn { __prg_replicate_6:n } #1
2763      { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
2764  \cs_new:cpn { __prg_replicate_7:n } #1
2765      { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1 }
2766  \cs_new:cpn { __prg_replicate_8:n } #1
2767      { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1 }
2768  \cs_new:cpn { __prg_replicate_9:n } #1
2769      { \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. . .

```
2770  \cs_new:cpn { __prg_replicate_first_-:n } #1
2771      {
2772        \exp_end:
2773        \__msg_kernel_expandable_error:nn { kernel } { negative-replication }
2774      }
2775  \cs_new:cpn { __prg_replicate_first_0:n } #1 { \exp_end: }
2776  \cs_new:cpn { __prg_replicate_first_1:n } #1 { \exp_end: #1 }
2777  \cs_new:cpn { __prg_replicate_first_2:n } #1 { \exp_end: #1#1 }
2778  \cs_new:cpn { __prg_replicate_first_3:n } #1 { \exp_end: #1#1#1 }
2779  \cs_new:cpn { __prg_replicate_first_4:n } #1 { \exp_end: #1#1#1#1 }
2780  \cs_new:cpn { __prg_replicate_first_5:n } #1 { \exp_end: #1#1#1#1#1 }
2781  \cs_new:cpn { __prg_replicate_first_6:n } #1 { \exp_end: #1#1#1#1#1#1 }
2782  \cs_new:cpn { __prg_replicate_first_7:n } #1 { \exp_end: #1#1#1#1#1#1#1 }
2783  \cs_new:cpn { __prg_replicate_first_8:n } #1 { \exp_end: #1#1#1#1#1#1#1#1 }
2784  \cs_new:cpn { __prg_replicate_first_9:n } #1 { \exp_end: #1#1#1#1#1#1#1#1#1 }
```

(*End definition for* `\prg_replicate:nn` *and others. These functions are documented on page 39.*)

5.7 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.

```
2785 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
2786    { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\mode_if_vertical:TF`*. This function is documented on page 40.*)

`\mode_if_horizontal_p:`
`\mode_if_horizontal:TF`

For testing horizontal mode.

```
2787 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
2788    { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\mode_if_horizontal:TF`*. This function is documented on page 39.*)

`\mode_if_inner_p:`
`\mode_if_inner:TF`

For testing inner mode.

```
2789 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
2790    { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\mode_if_inner:TF`*. This function is documented on page 40.*)

`\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.

```
2791 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
2792    { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\mode_if_math:TF`*. This function is documented on page 40.*)

5.8   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 issues 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 will get some sort of weird error message because the underlying \futurelet will store 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 code documented only in Appendix D of *The TEXbook*. . . 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.

```
2793 \cs_new:Npn \group_align_safe_begin:
2794    { \if_int_compare:w \if_false: { \fi: '} = \c_zero \fi: }
2795 \cs_new:Npn \group_align_safe_end:
2796    { \if_int_compare:w '{ = \c_zero } \fi: }
```

(*End definition for* `\group_align_safe_begin:` *and* `\group_align_safe_end:`*.*)

```
2797 ⟨@@=prg⟩
```

`\g__prg_map_int`   A nesting counter for mapping.

```
2798 \int_new:N \g__prg_map_int
```

(*End definition for* `\g__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 definition for* `\__prg_break_point:Nn` *and* `\__prg_map_break:Nn`.)

`\__prg_break_point:`
`\__prg_break:`
`\__prg_break:n`

Also done in l3basics as in format mode these are needed within l3alloc.

(*End definition for* `\__prg_break_point:` , `\__prg_break:` , *and* `\__prg_break:n`.)

5.9 Deprecated functions

`\scan_align_safe_stop:` Deprecated 2015-08-01 for removal after 2016-12-31.

```
2799 \cs_new_protected:Npn \scan_align_safe_stop: { }
```

(*End definition for* `\scan_align_safe_stop:`.)

```
2800 ⟨/initex | package⟩
```

6 **l3quark** implementation

The following test files are used for this code: `m3quark001.lvt`.

```
2801 ⟨*initex | package⟩
```

6.1 Quarks

```
2802 ⟨@@=quark⟩
```

`\quark_new:N` Allocate a new quark.

```
2803 \cs_new_protected:Npn \quark_new:N #1 { \tl_const:Nn #1 {#1} }
```

(*End definition for* `\quark_new:N`. *This function is documented on page 42.*)

`\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.

```
2804 \quark_new:N \q_nil
2805 \quark_new:N \q_mark
2806 \quark_new:N \q_no_value
2807 \quark_new:N \q_stop
```

(*End definition for* `\q_nil` *and others. These variables are documented on page 43.*)

`\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.

```
2808 \quark_new:N \q_recursion_tail
2809 \quark_new:N \q_recursion_stop
```

(*End definition for* `\q_recursion_tail` *and* `\q_recursion_stop`. *These variables are documented on page 44.*)

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.

```
2810 \cs_new:Npn \quark_if_recursion_tail_stop:N #1
2811   {
2812     \if_meaning:w \q_recursion_tail #1
2813       \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2814     \fi:
2815   }
2816 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
2817   {
2818     \if_meaning:w \q_recursion_tail #1
2819       \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
2820     \else:
2821       \exp_after:wN \use_none:n
2822     \fi:
2823   }
```

(*End definition for* `\quark_if_recursion_tail_stop:N` *and* `\quark_if_recursion_tail_stop_do:Nn`. *These functions are documented on page* *44*.)

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`.

```
2824 \cs_new:Npn \quark_if_recursion_tail_stop:n #1
2825   {
2826     \tl_if_empty:oTF
2827       { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
2828       { \use_none_delimit_by_q_recursion_stop:w }
2829       { }
2830   }
2831 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
2832   {
2833     \tl_if_empty:oTF
2834       { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
2835       { \use_i_delimit_by_q_recursion_stop:nw }
2836       { \use_none:n }
2837   }
2838 \cs_new:Npn \__quark_if_recursion_tail:w
2839     #1 \q_recursion_tail #2 ? #3 ?! { #1 #2 }
2840 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
2841 \cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn { o }
```

(*End 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* *44*.)

Analogs of the `\quark_if_recursion_tail_stop...` functions. Break the mapping using #2.

```
2842 \cs_new:Npn \__quark_if_recursion_tail_break:NN #1#2
2843   {
2844     \if_meaning:w \q_recursion_tail #1
```

304

```
2845        \exp_after:wN #2
2846      \fi:
2847    }
2848  \cs_new:Npn \__quark_if_recursion_tail_break:nN #1#2
2849    {
2850      \tl_if_empty:oTF
2851        { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
2852        {#2}
2853        { }
2854    }
```

(*End definition for* __quark_if_recursion_tail_break:NN *and* __quark_if_recursion_tail_break:nN.)

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.[7]

```
2855  \prg_new_conditional:Nnn \quark_if_nil:N { p, T , F , TF }
2856    {
2857      \if_meaning:w \q_nil #1
2858        \prg_return_true:
2859      \else:
2860        \prg_return_false:
2861      \fi:
2862    }
2863  \prg_new_conditional:Nnn \quark_if_no_value:N { p, T , F , TF }
2864    {
2865      \if_meaning:w \q_no_value #1
2866        \prg_return_true:
2867      \else:
2868        \prg_return_false:
2869      \fi:
2870    }
2871  \cs_generate_variant:Nn \quark_if_no_value_p:N { c }
2872  \cs_generate_variant:Nn \quark_if_no_value:NT  { c }
2873  \cs_generate_variant:Nn \quark_if_no_value:NF  { c }
2874  \cs_generate_variant:Nn \quark_if_no_value:NTF { c }
```

(*End definition for* \quark_if_nil:NTF *and* \quark_if_no_value:NTF. *These functions are documented on page 43.*)

Let us explain \quark_if_nil:n(TF). 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 the first ?! in the definition of \quark_if_nil:n stop #3.

```
2875  \prg_new_conditional:Nnn \quark_if_nil:n { p, T , F , TF }
2876    {
```

---

[7]It may still loop in special circumstances however!

305

```
2877        \__tl_if_empty_return:o
2878          { \__quark_if_nil:w {} #1 {} ? ! \q_nil ? ? ! }
2879      }
2880    \cs_new:Npn \__quark_if_nil:w #1 \q_nil #2 ? #3 ? ! { #1 #2 }
2881    \prg_new_conditional:Nnn \quark_if_no_value:n { p, T , F , TF }
2882      {
2883        \__tl_if_empty_return:o
2884          { \__quark_if_no_value:w {} #1 {} ? ! \q_no_value ? ? ! }
2885      }
2886    \cs_new:Npn \__quark_if_no_value:w #1 \q_no_value #2 ? #3 ? ! { #1 #2 }
2887    \cs_generate_variant:Nn \quark_if_nil_p:n { V , o }
2888    \cs_generate_variant:Nn \quark_if_nil:nTF { V , o }
2889    \cs_generate_variant:Nn \quark_if_nil:nT  { V , o }
2890    \cs_generate_variant:Nn \quark_if_nil:nF  { V , o }
```

(*End definition for* \quark_if_nil:nTF *and others. These functions are documented on page 43.*)

\q__tl_act_mark These private quarks are needed by l3tl, but that is loaded before the quark module,
\q__tl_act_stop hence their definition is deferred.

```
2891    \quark_new:N \q__tl_act_mark
2892    \quark_new:N \q__tl_act_stop
```

(*End definition for* \q__tl_act_mark *and* \q__tl_act_stop.)

6.2 Scan marks

```
2893    ⟨@@=scan⟩
```

\g__scan_marks_tl The list of all scan marks currently declared.

```
2894    \tl_new:N \g__scan_marks_tl
```

(*End definition for* \g__scan_marks_tl.)

__scan_new:N Check whether the variable is already a scan mark, then declare it to be equal to \scan_-
stop: globally.

```
2895    \cs_new_protected:Npn \__scan_new:N #1
2896      {
2897        \tl_if_in:NnTF \g__scan_marks_tl { #1 }
2898          {
2899            \__msg_kernel_error:nnx { kernel } { scanmark-already-defined }
2900              { \token_to_str:N #1 }
2901          }
2902          {
2903            \tl_gput_right:Nn \g__scan_marks_tl {#1}
2904            \cs_new_eq:NN #1 \scan_stop:
2905          }
2906      }
```

(*End definition for* __scan_new:N.)

\s__stop We only declare one scan mark here, more can be defined by specific modules.

```
2907    \__scan_new:N \s__stop
```

(*End definition for* \s__stop.)

__use_none_delimit_by_s__stop:w   Similar to \use_none_delimit_by_q_stop:w.

```
2908 \cs_new:Npn \__use_none_delimit_by_s__stop:w #1 \s__stop { }
```

(*End definition for* __use_none_delimit_by_s__stop:w.)

\s__seq   This private scan mark is needed by l3seq, but that is loaded before the quark module, hence its definition is deferred.

```
2909 \__scan_new:N \s__seq
```

(*End definition for* \s__seq.)

```
2910 ⟨/initex | package⟩
```

7   l3token implementation

```
2911 ⟨*initex | package⟩
```

```
2912 ⟨@@=char⟩
```

7.1   Manipulating and interrogating character tokens

\char_set_catcode:nn   Simple wrappers around the primitives.
\char_value_catcode:n
\char_show_value_catcode:n

```
2913 \cs_new_protected:Npn \char_set_catcode:nn #1#2
2914   {
2915     \tex_catcode:D \__int_eval:w #1 \__int_eval_end:
2916       = \__int_eval:w #2 \__int_eval_end:
2917   }
2918 \cs_new:Npn \char_value_catcode:n #1
2919   { \tex_the:D \tex_catcode:D \__int_eval:w #1\__int_eval_end: }
2920 \cs_new_protected:Npn \char_show_value_catcode:n #1
2921   { \__msg_show_wrap:n { > ~ \char_value_catcode:n {#1} } }
```

(*End definition for* \char_set_catcode:nn , \char_value_catcode:n , *and* \char_show_value_catcode:n. *These functions are documented on page 50.*)

\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

```
2922 \cs_new_protected:Npn \char_set_catcode_escape:N #1
2923   { \char_set_catcode:nn { `#1 } \c_zero }
2924 \cs_new_protected:Npn \char_set_catcode_group_begin:N #1
2925   { \char_set_catcode:nn { `#1 } \c_one }
2926 \cs_new_protected:Npn \char_set_catcode_group_end:N #1
2927   { \char_set_catcode:nn { `#1 } \c_two }
2928 \cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
2929   { \char_set_catcode:nn { `#1 } \c_three }
2930 \cs_new_protected:Npn \char_set_catcode_alignment:N #1
2931   { \char_set_catcode:nn { `#1 } \c_four }
2932 \cs_new_protected:Npn \char_set_catcode_end_line:N #1
2933   { \char_set_catcode:nn { `#1 } \c_five }
2934 \cs_new_protected:Npn \char_set_catcode_parameter:N #1
2935   { \char_set_catcode:nn { `#1 } \c_six }
2936 \cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
2937   { \char_set_catcode:nn { `#1 } \c_seven }
2938 \cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
2939   { \char_set_catcode:nn { `#1 } \c_eight }
2940 \cs_new_protected:Npn \char_set_catcode_ignore:N #1
```

```
2941      { \char_set_catcode:nn { `#1 } \c_nine }
2942    \cs_new_protected:Npn \char_set_catcode_space:N #1
2943      { \char_set_catcode:nn { `#1 } \c_ten }
2944    \cs_new_protected:Npn \char_set_catcode_letter:N #1
2945      { \char_set_catcode:nn { `#1 } \c_eleven }
2946    \cs_new_protected:Npn \char_set_catcode_other:N #1
2947      { \char_set_catcode:nn { `#1 } \c_twelve }
2948    \cs_new_protected:Npn \char_set_catcode_active:N #1
2949      { \char_set_catcode:nn { `#1 } \c_thirteen }
2950    \cs_new_protected:Npn \char_set_catcode_comment:N #1
2951      { \char_set_catcode:nn { `#1 } \c_fourteen }
2952    \cs_new_protected:Npn \char_set_catcode_invalid:N #1
2953      { \char_set_catcode:nn { `#1 } \c_fifteen }
```

(*End definition for* \char_set_catcode_escape:N *and others. These functions are documented on page 49.*)

<div style="float:left">

\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

</div>

```
2954    \cs_new_protected:Npn \char_set_catcode_escape:n #1
2955      { \char_set_catcode:nn {#1} \c_zero }
2956    \cs_new_protected:Npn \char_set_catcode_group_begin:n #1
2957      { \char_set_catcode:nn {#1} \c_one }
2958    \cs_new_protected:Npn \char_set_catcode_group_end:n #1
2959      { \char_set_catcode:nn {#1} \c_two }
2960    \cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
2961      { \char_set_catcode:nn {#1} \c_three }
2962    \cs_new_protected:Npn \char_set_catcode_alignment:n #1
2963      { \char_set_catcode:nn {#1} \c_four }
2964    \cs_new_protected:Npn \char_set_catcode_end_line:n #1
2965      { \char_set_catcode:nn {#1} \c_five }
2966    \cs_new_protected:Npn \char_set_catcode_parameter:n #1
2967      { \char_set_catcode:nn {#1} \c_six }
2968    \cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
2969      { \char_set_catcode:nn {#1} \c_seven }
2970    \cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
2971      { \char_set_catcode:nn {#1} \c_eight }
2972    \cs_new_protected:Npn \char_set_catcode_ignore:n #1
2973      { \char_set_catcode:nn {#1} \c_nine }
2974    \cs_new_protected:Npn \char_set_catcode_space:n #1
2975      { \char_set_catcode:nn {#1} \c_ten }
2976    \cs_new_protected:Npn \char_set_catcode_letter:n #1
2977      { \char_set_catcode:nn {#1} \c_eleven }
2978    \cs_new_protected:Npn \char_set_catcode_other:n #1
2979      { \char_set_catcode:nn {#1} \c_twelve }
2980    \cs_new_protected:Npn \char_set_catcode_active:n #1
2981      { \char_set_catcode:nn {#1} \c_thirteen }
2982    \cs_new_protected:Npn \char_set_catcode_comment:n #1
2983      { \char_set_catcode:nn {#1} \c_fourteen }
2984    \cs_new_protected:Npn \char_set_catcode_invalid:n #1
2985      { \char_set_catcode:nn {#1} \c_fifteen }
```

(*End definition for* \char_set_catcode_escape:n *and others. These functions are documented on page 49.*)

<div style="float:left">

\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

</div>

Pretty repetitive, but necessary!

```
2986 \cs_new_protected:Npn \char_set_mathcode:nn #1#2
2987   {
2988     \tex_mathcode:D \__int_eval:w #1 \__int_eval_end:
2989     = \__int_eval:w #2 \__int_eval_end:
2990   }
2991 \cs_new:Npn \char_value_mathcode:n #1
2992   { \tex_the:D \tex_mathcode:D \__int_eval:w #1\__int_eval_end: }
2993 \cs_new_protected:Npn \char_show_value_mathcode:n #1
2994   { \__msg_show_wrap:n { > ~ \char_value_mathcode:n {#1} } }
2995 \cs_new_protected:Npn \char_set_lccode:nn #1#2
2996   {
2997     \tex_lccode:D \__int_eval:w #1 \__int_eval_end:
2998     = \__int_eval:w #2 \__int_eval_end:
2999   }
3000 \cs_new:Npn \char_value_lccode:n #1
3001   { \tex_the:D \tex_lccode:D \__int_eval:w #1\__int_eval_end: }
3002 \cs_new_protected:Npn \char_show_value_lccode:n #1
3003   { \__msg_show_wrap:n { > ~ \char_value_lccode:n {#1} } }
3004 \cs_new_protected:Npn \char_set_uccode:nn #1#2
3005   {
3006     \tex_uccode:D \__int_eval:w #1 \__int_eval_end:
3007     = \__int_eval:w #2 \__int_eval_end:
3008   }
3009 \cs_new:Npn \char_value_uccode:n #1
3010   { \tex_the:D \tex_uccode:D \__int_eval:w #1\__int_eval_end: }
3011 \cs_new_protected:Npn \char_show_value_uccode:n #1
3012   { \__msg_show_wrap:n { > ~ \char_value_uccode:n {#1} } }
3013 \cs_new_protected:Npn \char_set_sfcode:nn #1#2
3014   {
3015     \tex_sfcode:D \__int_eval:w #1 \__int_eval_end:
3016     = \__int_eval:w #2 \__int_eval_end:
3017   }
3018 \cs_new:Npn \char_value_sfcode:n #1
3019   { \tex_the:D \tex_sfcode:D \__int_eval:w #1\__int_eval_end: }
3020 \cs_new_protected:Npn \char_show_value_sfcode:n #1
3021   { \__msg_show_wrap:n { > ~ \char_value_sfcode:n {#1} } }
```

(*End definition for* `\char_set_mathcode:nn` *and others. These functions are documented on page 51.*)

`\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`).

```
3022 \seq_new:N \l_char_special_seq
3023 \seq_set_split:Nnn \l_char_special_seq { }
3024   { \  \" \# \$ \% \& \\ \^ \_ \{ \} \~ }
3025 \seq_new:N \l_char_active_seq
3026 \seq_set_split:Nnn \l_char_special_seq { }
3027   { \" \$ \& \^ \_ \~ }
```

(*End definition for* `\l_char_active_seq` *and* `\l_char_special_seq`*. These variables are documented on page 51.*)

7.2 Creating character tokens

Four simple functions with very similar definitions, so set up using an auxiliary. These are similar to LuaTeX's `\letcharcode` primitive.

```
3028 \group_begin:
3029   \char_set_catcode_active:N \^^@
3030   \cs_set_protected:Npn \__char_tmp:nN #1#2
3031     {
3032       \cs_new_protected:cpn { #1 :nN } ##1
3033         {
3034           \group_begin:
3035             \char_set_catcode_active:n { ##1 }
3036             \char_set_lccode:nn { '\^^@ } { ##1 }
3037           \tex_lowercase:D { \group_end: #2 ^^@ }
3038         }
3039       \cs_new_protected:cpx { #1 :NN } ##1
3040         { \exp_not:c { #1 : nN } { '##1 } }
3041     }
3042   \__char_tmp:nN { char_set_active_eq }  \cs_set_eq:NN
3043   \__char_tmp:nN { char_gset_active_eq } \cs_gset_eq:NN
3044 \group_end:
3045 \cs_generate_variant:Nn \char_set_active_eq:NN  { Nc }
3046 \cs_generate_variant:Nn \char_gset_active_eq:NN { Nc }
3047 \cs_generate_variant:Nn \char_set_active_eq:nN  { nc }
3048 \cs_generate_variant:Nn \char_gset_active_eq:nN { nc }
```

(*End definition for* \char_set_active_eq:NN *and others. These functions are documented on page 47.*)

The aim here is to generate characters of (broadly) arbitrary category code. Where possible, that is done using engine support (XƎTeX, 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.

```
3049 \cs_new:Npn \char_generate:nn #1#2
3050   {
3051     \exp:w \exp_after:wN \__char_generate_aux:w
3052       \__int_value:w \__int_eval:w #1 \exp_after:wN ;
3053       \__int_value:w \__int_eval:w #2 ;
3054   }
3055 \cs_new:Npn \__char_generate:nn #1#2
3056   {
3057     \exp:w \exp_after:wN
3058       \__char_generate_aux:nnw \exp_after:wN
3059         { \__int_value:w \__int_eval:w #1 \exp_after:wN }
3060           {#2} \exp_end:
3061   }
```

Before doing any actual conversion, first some special case filtering. The `\Ucharcat` primitive cannot make active chars, so that is turned off here: if the primitive gets altered then the code is already in place for 8-bit engines and will kick in for LuaTeX too. Spaces are also banned 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 flagged up separately. That done, hand off to the engine-dependent part.

```
3062 \cs_new:Npn \__char_generate_aux:w #1 ; #2 ;
3063   {
```

```
3064        \if_int_compare:w #2 = \c_thirteen
3065          \__msg_kernel_expandable_error:nn { kernel } { char-active }
3066        \else:
3067          \if_int_compare:w #2 = \c_ten
3068            \if_int_compare:w #1 =  \c_zero
3069              \__msg_kernel_expandable_error:nn { kernel } { char-null-space }
3070            \else:
3071              \__msg_kernel_expandable_error:nn { kernel } { char-space }
3072            \fi:
3073          \else:
3074            \if_int_odd:w 0
3075                \if_int_compare:w #2 < \c_one     1 \fi:
3076                \if_int_compare:w #2 = \c_five    1 \fi:
3077                \if_int_compare:w #2 = \c_nine    1 \fi:
3078                \if_int_compare:w #2 > \c_thirteen 1 \fi: \exp_stop_f:
3079              \__msg_kernel_expandable_error:nn { kernel }
3080                { char-invalid-catcode }
3081            \else:
3082              \if_int_odd:w 0
3083                  \if_int_compare:w #1 < \c_zero        1 \fi:
3084                  \if_int_compare:w #1 > \c__char_max_int 1 \fi: \exp_stop_f:
3085                \__msg_kernel_expandable_error:nn { kernel }
3086                  { char-out-of-range }
3087              \else:
3088                \__char_generate_aux:nnw {#1} {#2}
3089              \fi:
3090            \fi:
3091          \fi:
3092        \fi:
3093        \exp_end:
3094      }
3095 \tl_new:N \l__char_tmp_tl
```

Engine-dependent definitions are now needed for the implementation. For LuaTeX and recent XeTeX releases there is engine-level support. They can do cases that macro emulation can't. All of those are filtered out here using a primitive-based boolean expression for speed. 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.

```
3096 \group_begin:
3097 ⟨*package⟩
3098   \char_set_catcode_active:N \^^L
3099   \cs_set:Npn ^^L { }
3100 ⟨/package⟩
3101   \char_set_catcode_other:n { 0 }
3102   \if_int_odd:w 0
3103      \cs_if_exist:NT \luatex_directlua:D { 1 }
3104      \cs_if_exist:NT \utex_charcat:D     { 1 } \exp_stop_f:
3105    \int_const:Nn \c__char_max_int { 1114111 }
3106    \cs_if_exist:NTF \luatex_directlua:D
3107      {
3108        \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
3109          {
3110              #3
3111              \exp_after:wN \exp_end:
```

```
3112              \luatex_directlua:D { l3kernel.charcat(#1, #2) }
3113            }
3114          }
3115          {
3116            \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
3117              {
3118                #3
3119                \exp_after:wN \exp_end:
3120                \utex_charcat:D  #1 ~ #2 ~
3121              }
3122          }
3123      \else:
```

For engines where `\Ucharcat` isn't available (or emulated) then 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. There are a few things to notice here. As `^^L` is `\outer` we need to locally set it to avoid a problem. To get open/close braces into the list, they are set up using `\if_false:` pairing and are then x-type expanded together into the desired form.

```
3124          \int_const:Nn \c__char_max_int { 255 }
3125          \tl_set:Nn \l__char_tmp_tl { \exp_not:N \or: }
3126          \char_set_catcode_group_begin:n { 0 } % {
3127          \tl_put_right:Nn \l__char_tmp_tl { ^^@ \if_false: } }
3128          \char_set_catcode_group_end:n { 0 }
3129          \tl_put_right:Nn \l__char_tmp_tl { { \fi: \exp_not:N \or: ^^@ } % }
3130          \tl_set:Nx \l__char_tmp_tl { \l__char_tmp_tl }
3131          \char_set_catcode_math_toggle:n { 0 }
3132          \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
```

As TeX will be very unhappy if if finds an alignment character inside a primitive `\halign` even when skipping false branches, some precautions are required. TeX will be happy if the token is hidden inside `\unexpanded` (which needs to be the primitive). The expansion chain here is required so that the conditional gets cleaned up correctly (other code assumes there is exactly one token to skip during the clean-up).

```
3133          \char_set_catcode_alignment:n { 0 }
3134          \tl_put_right:Nn \l__char_tmp_tl
3135            {
3136              \or:
3137                \etex_unexpanded:D \exp_after:wN
3138                  { \exp_after:wN ^^@ \exp_after:wN }
3139            }
3140          \tl_put_right:Nn \l__char_tmp_tl { \or: }
3141          \char_set_catcode_parameter:n { 0 }
3142          \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
3143          \char_set_catcode_math_superscript:n { 0 }
3144          \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
3145          \char_set_catcode_math_subscript:n { 0 }
3146          \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
3147          \tl_put_right:Nn \l__char_tmp_tl { \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. We also set up active tokens although they are (currently) filtered out by the interface layer (`\Ucharcat` cannot make active tokens).

```
3148        \char_set_catcode_space:n { 0 }
3149        \tl_put_right:No \l__char_tmp_tl { \use:n { \or: } ^^@ }
3150        \char_set_catcode_letter:n { 0 }
3151        \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
3152        \char_set_catcode_other:n { 0 }
3153        \tl_put_right:Nn \l__char_tmp_tl { \or: ^^@ }
3154        \char_set_catcode_active:n { 0 }
3155        \tl_put_right:Nn \l__char_tmp_tl { \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 x-type expansion ensures that `\tex_lowercase:D` receives the contents of the token list. In package mode, `^^L` is awkward hence this is done in three parts. Notice that at this stage `^^@` is active.

```
3156        \cs_set_protected:Npn \__char_tmp:n #1
3157          {
3158            \char_set_lccode:nn { 0 } {#1}
3159            \char_set_lccode:nn { 32 } {#1}
3160            \exp_args:Nx \tex_lowercase:D
3161              {
3162                \tl_const:Nn
3163                  \exp_not:c { c__char_ \__int_to_roman:w #1 _tl }
3164                  { \exp_not:o \l__char_tmp_tl }
3165              }
3166          }
3167 ⟨*package⟩
3168        \int_step_function:nnnN { 0 }  { 1 } { 11 }  \__char_tmp:n
3169        \group_begin:
3170          \tl_replace_once:Nnn \l__char_tmp_tl { ^^@ } { \ERROR }
3171          \__char_tmp:n { 12 }
3172        \group_end:
3173        \int_step_function:nnnN { 13 } { 1 } { 255 } \__char_tmp:n
3174 ⟨/package⟩
3175 ⟨*initex⟩
3176        \int_step_function:nnnN { 0 }  { 1 } { 255 }  \__char_tmp:n
3177 ⟨/initex⟩
3178        \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
3179          {
3180            #3
3181            \exp_after:wN \exp_after:wN
3182            \exp_after:wN \exp_end:
3183            \exp_after:wN \exp_after:wN
3184            \if_case:w #2
3185              \exp_last_unbraced:Nv \exp_stop_f:
3186                { c__char_ \__int_to_roman:w #1 _tl }
3187            \fi:
3188          }
3189    \fi:
3190 \group_end:
```

(*End definition for* `\char_generate:nn` *and others. These functions are documented on page 48.*)

7.3   Generic tokens

```
3191 ⟨@@=token⟩
```

313

These are all defined in l3basics, as they are needed "early". This is just a reminder!

(*End definition for* `\token_to_meaning:N` *and* `\token_to_str:N`*. These functions are documented on page 52.*)

Creates a new token.

```
3192 \cs_new_protected:Npn \token_new:Nn #1#2 { \cs_new_eq:NN #1 #2 }
```

(*End definition for* `\token_new:Nn`*. This function is documented on page 52.*)

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 `\__chk_if_free_cs:N` check.

```
3193 \group_begin:
3194   \__chk_if_free_cs:N \c_group_begin_token
3195   \tex_global:D \tex_let:D \c_group_begin_token {
3196   \__chk_if_free_cs:N \c_group_end_token
3197   \tex_global:D \tex_let:D \c_group_end_token }
3198   \char_set_catcode_math_toggle:N \*
3199   \cs_new_eq:NN \c_math_toggle_token *
3200   \char_set_catcode_alignment:N \*
3201   \cs_new_eq:NN \c_alignment_token *
3202   \cs_new_eq:NN \c_parameter_token #
3203   \cs_new_eq:NN \c_math_superscript_token ^
3204   \char_set_catcode_math_subscript:N \*
3205   \cs_new_eq:NN \c_math_subscript_token *
3206   \__chk_if_free_cs:N \c_space_token
3207   \use:n { \tex_global:D \tex_let:D \c_space_token = ~ } ~
3208   \cs_new_eq:NN \c_catcode_letter_token a
3209   \cs_new_eq:NN \c_catcode_other_token 1
3210 \group_end:
```

(*End definition for* `\c_group_begin_token` *and others. These functions are documented on page 52.*)

Not an implicit token!

```
3211 \group_begin:
3212   \char_set_catcode_active:N \*
3213   \tl_const:Nn \c_catcode_active_tl { \exp_not:N * }
3214 \group_end:
```

(*End definition for* `\c_catcode_active_tl`*. This variable is documented on page 52.*)

7.4 Token conditionals

Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.

```
3215 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
3216   {
3217     \if_catcode:w \exp_not:N #1 \c_group_begin_token
3218       \prg_return_true: \else: \prg_return_false: \fi:
3219   }
```

(*End definition for* `\token_if_group_begin:NTF`. *This function is documented on page 53.*)

`\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.

```
3220 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
3221   {
3222     \if_catcode:w \exp_not:N #1 \c_group_end_token
3223       \prg_return_true: \else: \prg_return_false: \fi:
3224   }
```

(*End definition for* `\token_if_group_end:NTF`. *This function is documented on page 53.*)

`\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.

```
3225 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
3226   {
3227     \if_catcode:w \exp_not:N #1 \c_math_toggle_token
3228       \prg_return_true: \else: \prg_return_false: \fi:
3229   }
```

(*End definition for* `\token_if_math_toggle:NTF`. *This function is documented on page 53.*)

`\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.

```
3230 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
3231   {
3232     \if_catcode:w \exp_not:N #1 \c_alignment_token
3233       \prg_return_true: \else: \prg_return_false: \fi:
3234   }
```

(*End definition for* `\token_if_alignment:NTF`. *This function is documented on page 53.*)

`\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 will remain after the group.

```
3235 \group_begin:
3236 \cs_set_eq:NN \c_parameter_token \scan_stop:
3237 \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
3238   {
3239     \if_catcode:w \exp_not:N #1 \c_parameter_token
3240       \prg_return_true: \else: \prg_return_false: \fi:
3241   }
3242 \group_end:
```

(*End definition for* `\token_if_parameter:NTF`. *This function is documented on page 53.*)

`\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.

```
3243 \prg_new_conditional:Npnn \token_if_math_superscript:N #1
3244   { p , T , F , TF }
3245   {
3246     \if_catcode:w \exp_not:N #1 \c_math_superscript_token
3247       \prg_return_true: \else: \prg_return_false: \fi:
3248   }
```

(End definition for `\token_if_math_superscript:NTF`. *This function is documented on page 53.)*

`\token_if_math_subscript_p:N`
`\token_if_math_subscript:N`*TF*

Check if token is a math subscript token. We use the constant `\c_math_subscript_-`
`token` for this.

```
3249 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T ,  F , TF }
3250   {
3251     \if_catcode:w \exp_not:N #1 \c_math_subscript_token
3252       \prg_return_true: \else: \prg_return_false: \fi:
3253   }
```

(End definition for `\token_if_math_subscript:NTF`. *This function is documented on page 53.)*

`\token_if_space_p:N`
`\token_if_space:N`*TF*

Check if token is a space token. We use the constant `\c_space_token` for this.

```
3254 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T ,  F , TF }
3255   {
3256     \if_catcode:w \exp_not:N #1 \c_space_token
3257       \prg_return_true: \else: \prg_return_false: \fi:
3258   }
```

(End definition for `\token_if_space:NTF`. *This function is documented on page 53.)*

`\token_if_letter_p:N`
`\token_if_letter:N`*TF*

Check if token is a letter token. We use the constant `\c_catcode_letter_token` for this.

```
3259 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T ,  F , TF }
3260   {
3261     \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
3262       \prg_return_true: \else: \prg_return_false: \fi:
3263   }
```

(End definition for `\token_if_letter:NTF`. *This function is documented on page 54.)*

`\token_if_other_p:N`
`\token_if_other:N`*TF*

Check if token is an other char token. We use the constant `\c_catcode_other_token`
for this.

```
3264 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T ,  F , TF }
3265   {
3266     \if_catcode:w \exp_not:N #1 \c_catcode_other_token
3267       \prg_return_true: \else: \prg_return_false: \fi:
3268   }
```

(End definition for `\token_if_other:NTF`. *This function is documented on page 54.)*

`\token_if_active_p:N`
`\token_if_active:N`*TF*

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.

```
3269 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T ,  F , TF }
3270   {
3271     \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
3272       \prg_return_true: \else: \prg_return_false: \fi:
3273   }
```

(End definition for `\token_if_active:NTF`. *This function is documented on page 54.)*

Check if the tokens `#1` and `#2` have same meaning.

```
3274 \prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T , F , TF }
3275   {
3276     \if_meaning:w  #1  #2
3277       \prg_return_true: \else: \prg_return_false: \fi:
3278   }
```

(*End definition for* `\token_if_eq_meaning:NNTF`*. This function is documented on page 54.*)

Check if the tokens `#1` and `#2` have same category code.

```
3279 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T ,  F , TF }
3280   {
3281     \if_catcode:w \exp_not:N #1 \exp_not:N #2
3282       \prg_return_true: \else: \prg_return_false: \fi:
3283   }
```

(*End definition for* `\token_if_eq_catcode:NNTF`*. This function is documented on page 54.*)

Check if the tokens `#1` and `#2` have same character code.

```
3284 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T ,  F , TF }
3285   {
3286     \if_charcode:w \exp_not:N #1 \exp_not:N #2
3287       \prg_return_true: \else: \prg_return_false: \fi:
3288   }
```

(*End definition for* `\token_if_eq_charcode:NNTF`*. This function is documented on page 54.*)

When a token is a macro, `\token_to_meaning:N` will always output 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.

```
3289 \use:x
3290   {
3291     \prg_new_conditional:Npnn \exp_not:N \token_if_macro:N ##1
3292       { p , T ,  F , TF }
3293       {
3294         \exp_not:N \exp_after:wN \exp_not:N \__token_if_macro_p:w
3295         \exp_not:N \token_to_meaning:N ##1 \tl_to_str:n { ma : }
3296           \exp_not:N \q_stop
3297       }
3298     \cs_new:Npn \exp_not:N  \__token_if_macro_p:w
3299       ##1 \tl_to_str:n { ma } ##2 \c_colon_str ##3 \exp_not:N \q_stop
3300   }
```

317

```
3301        {
3302          \if_int_compare:w \__str_if_eq_x:nn { #2 } { cro } = \c_zero
3303              \prg_return_true:
3304          \else:
3305              \prg_return_false:
3306          \fi:
3307        }
```

(*End definition for* `\token_if_macro:NTF` *and* `\__token_if_macro_p:w`*. These functions are documented on page 54.*)

`\token_if_cs_p:N` Check if token has same catcode as a control sequence. This follows the same pattern as
`\token_if_cs:NTF` for `\token_if_letter:N` *etc.* We use `\scan_stop:` for this.

```
3308 \prg_new_conditional:Npnn \token_if_cs:N #1 { p , T ,  F , TF }
3309   {
3310     \if_catcode:w \exp_not:N #1 \scan_stop:
3311        \prg_return_true: \else: \prg_return_false: \fi:
3312   }
```

(*End definition for* `\token_if_cs:NTF`*. This function is documented on page 54.*)

`\token_if_expandable_p:N`   Check if token is expandable. We use the fact that TEX will temporarily convert `\exp_-`
`\token_if_expandable:NTF` `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).

```
3313 \prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T ,  F , TF }
3314   {
3315     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
3316        \prg_return_false:
3317     \else:
3318        \if_cs_exist:N #1
3319          \prg_return_true:
3320        \else:
3321          \prg_return_false:
3322        \fi:
3323     \fi:
3324   }
```

(*End definition for* `\token_if_expandable:NTF`*. This function is documented on page 54.*)

`\__token_delimit_by_char":w`   These auxiliary functions are used below to define some conditionals which detect whether
`\__token_delimit_by_count:w` the `\meaning` of their argument begins with a particular string. Each auxiliary takes an
`\__token_delimit_by_dimen:w` argument delimited by a string, a second one delimited by `\q_stop`, and returns the first
`\__token_delimit_by_macro:w` one and its delimiter. This result will eventually be compared to another string.
`\__token_delimit_by_muskip:w`
`\__token_delimit_by_skip:w`
`\__token_delimit_by_toks:w`

```
3325 \group_begin:
3326 \cs_set_protected:Npn \__token_tmp:w #1
3327   {
3328     \use:x
3329       {
3330         \cs_new:Npn \exp_not:c { __token_delimit_by_ #1 :w }
3331             ####1 \tl_to_str:n {#1} ####2 \exp_not:N \q_stop
3332           { ####1 \tl_to_str:n {#1} }
3333       }
3334   }
```

318

```
3335    \__token_tmp:w { char" }
3336    \__token_tmp:w { count }
3337    \__token_tmp:w { dimen }
3338    \__token_tmp:w { macro }
3339    \__token_tmp:w { muskip }
3340    \__token_tmp:w { skip }
3341    \__token_tmp:w { toks }
3342  \group_end:
```

(*End 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_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_x_return:nn 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 x-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 five conditionals, \cs_if_exist:cT turns out to be false, 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).

```
3343  \group_begin:
3344  \cs_set_protected:Npn \__token_tmp:w #1#2#3
3345    {
3346      \use:x
3347        {
3348          \prg_new_conditional:Npnn \exp_not:c { token_if_ #1 :N } ####1
3349            { p , T , F , TF }
3350            {
3351              \cs_if_exist:cT { tex_ #2 :D }
3352                {
3353                  \exp_not:N \if_meaning:w ####1 \exp_not:c { tex_ #2 :D }
3354                  \exp_not:N \prg_return_false:
3355                  \exp_not:N \else:
```

319

```
3356                    \exp_not:N \if_meaning:w ####1 \exp_not:c { tex_ #2 def:D }
3357                    \exp_not:N \prg_return_false:
3358                    \exp_not:N \else:
3359                  }
3360              \exp_not:N \__str_if_eq_x_return:nn
3361                {
3362                  \exp_not:N \exp_after:wN
3363                  \exp_not:c { __token_delimit_by_ #2 :w }
3364                  \exp_not:N \token_to_meaning:N ####1
3365                  ? \tl_to_str:n {#2} \exp_not:N \q_stop
3366                }
3367                { \exp_not:n {#3} }
3368              \cs_if_exist:cT { tex_ #2 :D }
3369                {
3370                  \exp_not:N \fi:
3371                  \exp_not:N \fi:
3372                }
3373          }
3374      }
3375  }
3376 \__token_tmp:w { chardef } { char" } { \token_to_str:N \char" }
3377 \__token_tmp:w { mathchardef } { char" } { \token_to_str:N \mathchar" }
3378 \__token_tmp:w { long_macro } { macro } { \tl_to_str:n { \long } macro }
3379 \__token_tmp:w { protected_macro } { macro }
3380   { \tl_to_str:n { \protected } macro }
3381 \__token_tmp:w { protected_long_macro } { macro }
3382   { \token_to_str:N \protected \tl_to_str:n { \long } macro }
3383 \__token_tmp:w { dim_register } { dimen } { \token_to_str:N \dimen }
3384 \__token_tmp:w { int_register } { count } { \token_to_str:N \count }
3385 \__token_tmp:w { muskip_register } { muskip } { \token_to_str:N \muskip }
3386 \__token_tmp:w { skip_register } { skip } { \token_to_str:N \skip }
3387 \__token_tmp:w { toks_register } { toks } { \token_to_str:N \toks }
3388 \group_end:
```

(*End definition for* `\token_if_chardef:NTF` *and others. These functions are documented on page 55.*)

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 inexistent 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 `\q_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 excep-

320

tions 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.

```
3389 \tex_chardef:D \c__token_A_int = ‘A ~ %
3390 \use:x
3391   {
3392     \prg_new_conditional:Npnn \exp_not:N \token_if_primitive:N ##1
3393       { p , T , F , TF }
3394       {
3395         \exp_not:N \token_if_macro:NTF ##1
3396           \exp_not:N \prg_return_false:
3397           {
3398             \exp_not:N \exp_after:wN \exp_not:N \__token_if_primitive:NNw
3399             \exp_not:N \token_to_meaning:N ##1
3400               \tl_to_str:n { : : : } \exp_not:N \q_stop ##1
3401           }
3402       }
3403     \cs_new:Npn \exp_not:N \__token_if_primitive:NNw
3404       ##1##2 ##3 \c_colon_str ##4 \exp_not:N \q_stop
3405       {
3406         \exp_not:N \tl_if_empty:oTF
3407           { \exp_not:N \__token_if_primitive_space:w ##3 ~ }
3408           {
3409             \exp_not:N \__token_if_primitive_loop:N ##3
3410               \c_colon_str \exp_not:N \q_stop
3411           }
3412           { \exp_not:N \__token_if_primitive_nullfont:N }
3413       }
3414   }
3415 \cs_new:Npn \__token_if_primitive_space:w #1 ~ { }
3416 \cs_new:Npn \__token_if_primitive_nullfont:N #1
3417   {
3418     \if_meaning:w \tex_nullfont:D #1
3419       \prg_return_true:
3420     \else:
3421       \prg_return_false:
3422     \fi:
3423   }
3424 \cs_new:Npn \__token_if_primitive_loop:N #1
3425   {
3426     \if_int_compare:w ‘#1 < \c__token_A_int %
3427       \exp_after:wN \__token_if_primitive:Nw
3428       \exp_after:wN #1
3429     \else:
3430       \exp_after:wN \__token_if_primitive_loop:N
3431     \fi:
3432   }
3433 \cs_new:Npn \__token_if_primitive:Nw #1 #2 \q_stop
3434   {
```

```
3435    \if:w : #1
3436      \exp_after:wN \__token_if_primitive_undefined:N
3437    \else:
3438      \prg_return_false:
3439      \exp_after:wN \use_none:n
3440    \fi:
3441  }
3442 \cs_new:Npn \__token_if_primitive_undefined:N #1
3443  {
3444    \if_cs_exist:N #1
3445      \prg_return_true:
3446    \else:
3447      \prg_return_false:
3448    \fi:
3449  }
```

(*End definition for* `\token_if_primitive:NTF` *and others. These functions are documented on page 56.*)

7.5   Peeking ahead at the next token

```
3450 ⟨@@=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.

```
3451 \cs_new_eq:NN \l_peek_token ?
3452 \cs_new_eq:NN \g_peek_token ?
```

(*End definition for* `\l_peek_token` *and* `\g_peek_token`. *These variables are documented on page 56.*)

`\l__peek_search_token`

The token to search for as an implicit token: *cf.* `\l__peek_search_tl`.

```
3453 \cs_new_eq:NN \l__peek_search_token ?
```

(*End definition for* `\l__peek_search_token`.)

`\l__peek_search_tl`

The token to search for as an explicit token: *cf.* `\l__peek_search_token`.

```
3454 \tl_new:N \l__peek_search_tl
```

(*End 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.

```
3455 \cs_new:Npn \__peek_true:w  { }
3456 \cs_new:Npn \__peek_true_aux:w  { }
3457 \cs_new:Npn \__peek_false:w { }
3458 \cs_new:Npn \__peek_tmp:w { }
```

(*End definition for* `\__peek_true:w` *and others.*)

`\peek_after:Nw`
`\peek_gafter:Nw` Simple wrappers for `\futurelet`: no arguments absorbed here.

```
3459 \cs_new_protected:Npn \peek_after:Nw
3460   { \tex_futurelet:D \l_peek_token }
3461 \cs_new_protected:Npn \peek_gafter:Nw
3462   { \tex_global:D \tex_futurelet:D \g_peek_token }
```

(*End definition for* `\peek_after:Nw` *and* `\peek_gafter:Nw`. *These functions are documented on page 56*.)

`\__peek_true_remove:w` A function to remove the next token and then regain control.

```
3463 \cs_new_protected:Npn \__peek_true_remove:w
3464   {
3465     \group_align_safe_end:
3466     \tex_afterassignment:D \__peek_true_aux:w
3467     \cs_set_eq:NN \__peek_tmp:w
3468   }
```

(*End definition for* `\__peek_true_remove:w`.)

`\__peek_token_generic:NNTF` The generic function stores 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.

```
3469 \cs_new_protected:Npn \__peek_token_generic:NNTF #1#2#3#4
3470   {
3471     \cs_set_eq:NN \l__peek_search_token #2
3472     \tl_set:Nn \l__peek_search_tl {#2}
3473     \cs_set:Npx \__peek_true:w
3474       {
3475         \exp_not:N \group_align_safe_end:
3476         \exp_not:n {#3}
3477       }
3478     \cs_set:Npx \__peek_false:w
3479       {
3480         \exp_not:N \group_align_safe_end:
3481         \exp_not:n {#4}
3482       }
3483     \group_align_safe_begin:
3484       \peek_after:Nw #1
3485   }
3486 \cs_new_protected:Npn \__peek_token_generic:NNT #1#2#3
3487   { \__peek_token_generic:NNTF #1 #2 {#3} { } }
3488 \cs_new_protected:Npn \__peek_token_generic:NNF #1#2#3
3489   { \__peek_token_generic:NNTF #1 #2 { } {#3} }
```

(*End definition for* `\__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.

```
3490 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF #1#2#3#4
3491   {
3492     \cs_set_eq:NN \l__peek_search_token #2
3493     \tl_set:Nn \l__peek_search_tl {#2}
3494     \cs_set_eq:NN \__peek_true:w \__peek_true_remove:w
3495     \cs_set:Npx \__peek_true_aux:w { \exp_not:n {#3} }
3496     \cs_set:Npx \__peek_false:w
```

```
3497        {
3498          \exp_not:N \group_align_safe_end:
3499          \exp_not:n {#4}
3500        }
3501      \group_align_safe_begin:
3502        \peek_after:Nw #1
3503    }
3504  \cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
3505    { \__peek_token_remove_generic:NNTF #1 #2 {#3} { } }
3506  \cs_new_protected:Npn \__peek_token_remove_generic:NNF #1#2#3
3507    { \__peek_token_remove_generic:NNTF #1 #2 { } {#3} }
```

(*End definition for* __peek_token_remove_generic:NNTF.)

__peek_execute_branches_meaning: The meaning test is straight forward.

```
3508  \cs_new:Npn \__peek_execute_branches_meaning:
3509    {
3510      \if_meaning:w \l_peek_token \l__peek_search_token
3511        \exp_after:wN \__peek_true:w
3512      \else:
3513        \exp_after:wN \__peek_false:w
3514      \fi:
3515    }
```

(*End 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 will only be 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.

```
3516  \cs_new:Npn \__peek_execute_branches_catcode:
3517    { \if_catcode:w \__peek_execute_branches_catcode_aux: }
3518  \cs_new:Npn \__peek_execute_branches_charcode:
3519    { \if_charcode:w \__peek_execute_branches_catcode_aux: }
3520  \cs_new:Npn \__peek_execute_branches_catcode_aux:
3521    {
```

```
3522          \if_catcode:w \exp_not:N \l_peek_token \scan_stop:
3523            \exp_after:wN \exp_after:wN
3524            \exp_after:wN \__peek_execute_branches_catcode_auxii:N
3525            \exp_after:wN \exp_not:N
3526          \else:
3527            \exp_after:wN \__peek_execute_branches_catcode_auxiii:
3528          \fi:
3529      }
3530    \cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1
3531      {
3532            \exp_not:N #1
3533            \exp_after:wN \exp_not:N \l__peek_search_tl
3534          \exp_after:wN \__peek_true:w
3535        \else:
3536          \exp_after:wN \__peek_false:w
3537        \fi:
3538        #1
3539      }
3540    \cs_new:Npn \__peek_execute_branches_catcode_auxiii:
3541      {
3542            \exp_not:N \l_peek_token
3543            \exp_after:wN \exp_not:N \l__peek_search_tl
3544          \exp_after:wN \__peek_true:w
3545        \else:
3546          \exp_after:wN \__peek_false:w
3547        \fi:
3548      }
```

(*End definition for* `\__peek_execute_branches_catcode:` *and others.*)

`\__peek_ignore_spaces_execute_branches:` This function removes one space token at a time, and calls `\__peek_execute_branches:` when encountering the first non-space token. We directly use the primitive meaning test rather than `\token_if_eq_meaning:NNTF` because `\l_peek_token` may be an outer macro (coming from non-LATEX3 packages). Spaces are removed using a side-effect of f-expansion: `\exp:w \exp_end_continue_f:w` removes one space.

```
3549    \cs_new_protected:Npn \__peek_ignore_spaces_execute_branches:
3550      {
3551        \if_meaning:w \l_peek_token \c_space_token
3552          \exp_after:wN \peek_after:Nw
3553          \exp_after:wN \__peek_ignore_spaces_execute_branches:
3554          \exp:w \exp_end_continue_f:w
3555        \else:
3556          \exp_after:wN \__peek_execute_branches:
3557        \fi:
3558      }
```

(*End definition for* `\__peek_ignore_spaces_execute_branches:`.)

`\__peek_def:nnnn`
`\__peek_def:nnnnn` The public functions themselves cannot be defined using `\prg_new_conditional:Npnn` and so a couple of auxiliary functions are used. As a result, everything is done inside a group. As a result things are a bit complicated.

```
3559    \group_begin:
3560      \cs_set:Npn \__peek_def:nnnn #1#2#3#4
3561        {
```

```
3562          \__peek_def:nnnnn {#1} {#2} {#3} {#4} { TF }
3563          \__peek_def:nnnnn {#1} {#2} {#3} {#4} { T }
3564          \__peek_def:nnnnn {#1} {#2} {#3} {#4} { F }
3565       }
3566    \cs_set:Npn \__peek_def:nnnnn #1#2#3#4#5
3567       {
3568          \cs_new_protected:cpx { #1 #5 }
3569             {
3570                \tl_if_empty:nF {#2}
3571                   { \exp_not:n { \cs_set_eq:NN \__peek_execute_branches: #2 } }
3572                \exp_not:c { #3 #5 }
3573                \exp_not:n {#4}
3574             }
3575       }
```

(*End definition for* `\__peek_def:nnnn` *and* `\__peek_def:nnnnn`.)

eek_catcode:NTF

With everything in place the definitions can take place. First for category codes.

```
3576    \__peek_def:nnnn { peek_catcode:N }
3577       { }
3578       { __peek_token_generic:NN }
3579       { \__peek_execute_branches_catcode: }
3580    \__peek_def:nnnn { peek_catcode_ignore_spaces:N }
3581       { \__peek_execute_branches_catcode: }
3582       { __peek_token_generic:NN }
3583       { \__peek_ignore_spaces_execute_branches: }
3584    \__peek_def:nnnn { peek_catcode_remove:N }
3585       { }
3586       { __peek_token_remove_generic:NN }
3587       { \__peek_execute_branches_catcode: }
3588    \__peek_def:nnnn { peek_catcode_remove_ignore_spaces:N }
3589       { \__peek_execute_branches_catcode: }
3590       { __peek_token_remove_generic:NN }
3591       { \__peek_ignore_spaces_execute_branches: }
```

(*End definition for* `\peek_catcode:NTF` *and others. These functions are documented on page 56.*)

Then for character codes.

```
3592    \__peek_def:nnnn { peek_charcode:N }
3593       { }
3594       { __peek_token_generic:NN }
3595       { \__peek_execute_branches_charcode: }
3596    \__peek_def:nnnn { peek_charcode_ignore_spaces:N }
3597       { \__peek_execute_branches_charcode: }
3598       { __peek_token_generic:NN }
3599       { \__peek_ignore_spaces_execute_branches: }
3600    \__peek_def:nnnn { peek_charcode_remove:N }
3601       { }
3602       { __peek_token_remove_generic:NN }
3603       { \__peek_execute_branches_charcode: }
3604    \__peek_def:nnnn { peek_charcode_remove_ignore_spaces:N }
3605       { \__peek_execute_branches_charcode: }
3606       { __peek_token_remove_generic:NN }
3607       { \__peek_ignore_spaces_execute_branches: }
```

(End definition for `\peek_charcode:NTF` *and others. These functions are documented on page 57.)*

`\peek_meaning:N`*TF*
`\peek_meaning_ignore_spaces:N`*TF*
`\peek_meaning_remove:N`*TF*
`\peek_meaning_remove_ignore_spaces:N`*TF*

Finally for meaning, with the group closed to remove the temporary definition functions.

```
3608    \__peek_def:nnnn { peek_meaning:N }
3609      { }
3610      { __peek_token_generic:NN }
3611      { \__peek_execute_branches_meaning: }
3612    \__peek_def:nnnn { peek_meaning_ignore_spaces:N }
3613      { \__peek_execute_branches_meaning: }
3614      { __peek_token_generic:NN }
3615      { \__peek_ignore_spaces_execute_branches: }
3616    \__peek_def:nnnn { peek_meaning_remove:N }
3617      { }
3618      { __peek_token_remove_generic:NN }
3619      { \__peek_execute_branches_meaning: }
3620    \__peek_def:nnnn { peek_meaning_remove_ignore_spaces:N }
3621      { \__peek_execute_branches_meaning: }
3622      { __peek_token_remove_generic:NN }
3623      { \__peek_ignore_spaces_execute_branches: }
3624    \group_end:
```

(End definition for `\peek_meaning:NTF` *and others. These functions are documented on page 58.)*

7.6   Decomposing a macro definition

`\token_get_prefix_spec:N`
`\token_get_arg_spec:N`
`\token_get_replacement_spec:N`
`\__peek_get_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 argument 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.

```
3625    \exp_args:Nno \use:nn
3626      { \cs_new:Npn \__peek_get_prefix_arg_replacement:wN #1 }
3627      { \tl_to_str:n { macro : } #2 -> #3 \q_stop #4 }
3628      { #4 {#1} {#2} {#3} }
3629    \cs_new:Npn \token_get_prefix_spec:N #1
3630      {
3631        \token_if_macro:NTF #1
3632          {
3633            \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3634              \token_to_meaning:N #1 \q_stop \use_i:nnn
3635          }
3636          { \scan_stop: }
3637      }
3638    \cs_new:Npn \token_get_arg_spec:N #1
3639      {
3640        \token_if_macro:NTF #1
3641          {
3642            \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3643              \token_to_meaning:N #1 \q_stop \use_ii:nnn
3644          }
3645          { \scan_stop: }
3646      }
```

```
3647 \cs_new:Npn \token_get_replacement_spec:N #1
3648   {
3649     \token_if_macro:NTF #1
3650       {
3651         \exp_after:wN \__peek_get_prefix_arg_replacement:wN
3652           \token_to_meaning:N #1 \q_stop \use_iii:nnn
3653       }
3654       { \scan_stop: }
3655   }
```

(*End definition for* `\token_get_prefix_spec:N` *and others. These functions are documented on page 59.*)

```
3656 ⟨/initex | package⟩
```

8 **l3int** implementation

```
3657 ⟨*initex | package⟩
```

```
3658 ⟨@@=int⟩
```

The following test files are used for this code: **m3int001,m3int002,m3int03.**

`\c_max_register_int` Done in l3basics.

(*End definition for* `\c_max_register_int`. *This variable is documented on page 72.*)

`\__int_to_roman:w` Done in l3basics.
`\if_int_compare:w`

(*End definition for* `\__int_to_roman:w` *and* `\if_int_compare:w`.)

`\or:` Done in l3basics.

(*End definition for* `\or:`. *This function is documented on page 73.*)

`\__int_value:w` Here are the remaining primitives for number comparisons and expressions.
`\__int_eval:w`
`\__int_eval_end:`
`\if_int_odd:w`
`\if_case:w`

```
3659 \cs_new_eq:NN \__int_value:w       \tex_number:D
3660 \cs_new_eq:NN \__int_eval:w        \etex_numexpr:D
3661 \cs_new_eq:NN \__int_eval_end:     \tex_relax:D
3662 \cs_new_eq:NN \if_int_odd:w        \tex_ifodd:D
3663 \cs_new_eq:NN \if_case:w           \tex_ifcase:D
```

(*End definition for* `\__int_value:w` *and others.*)

8.1 Integer expressions

`\int_eval:n` Wrapper for `\__int_eval:w`. Can be used in an integer expression or directly in the input stream. In format mode, there is already a definition in l3alloc for bootstrapping, which is therefore corrected to the "real" version here.

```
3664 ⟨*initex⟩
3665 \cs_set:Npn \int_eval:n #1
3666   { \__int_value:w \__int_eval:w #1 \__int_eval_end: }
3667 ⟨/initex⟩
3668 ⟨*package⟩
3669 \cs_new:Npn \int_eval:n #1
3670   { \__int_value:w \__int_eval:w #1 \__int_eval_end: }
3671 ⟨/package⟩
```

(*End definition for* `\int_eval:n`. *This function is documented on page 62.*)

`\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.

```
3672 \cs_new:Npn \int_abs:n #1
3673   {
3674     \__int_value:w \exp_after:wN \__int_abs:N
3675       \__int_value:w \__int_eval:w #1 \__int_eval_end:
3676     \exp_stop_f:
3677   }
3678 \cs_new:Npn \__int_abs:N #1
3679   { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
3680 \cs_set:Npn \int_max:nn #1#2
3681   {
3682     \__int_value:w \exp_after:wN \__int_maxmin:wwN
3683       \__int_value:w \__int_eval:w #1 \exp_after:wN ;
3684       \__int_value:w \__int_eval:w #2 ;
3685       >
3686     \exp_stop_f:
3687   }
3688 \cs_set:Npn \int_min:nn #1#2
3689   {
3690     \__int_value:w \exp_after:wN \__int_maxmin:wwN
3691       \__int_value:w \__int_eval:w #1 \exp_after:wN ;
3692       \__int_value:w \__int_eval:w #2 ;
3693       <
3694     \exp_stop_f:
3695   }
3696 \cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
3697   {
3698     \if_int_compare:w #1 #3 #2 ~
3699       #1
3700     \else:
3701       #2
3702     \fi:
3703   }
```

(*End definition for* `\int_abs:n` *and others. These functions are documented on page 62.*)

`\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 $(|#3#4|-1)/2$, 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.

```
3704 \cs_new:Npn \int_div_truncate:nn #1#2
3705   {
3706     \__int_value:w \__int_eval:w
3707       \exp_after:wN \__int_div_truncate:NwNw
3708       \__int_value:w \__int_eval:w #1 \exp_after:wN ;
```

329

```
3709        \__int_value:w \__int_eval:w #2 ;
3710      \__int_eval_end:
3711    }
3712  \cs_new:Npn \__int_div_truncate:NwNw #1#2; #3#4;
3713    {
3714      \if_meaning:w 0 #1
3715        \c_zero
3716      \else:
3717        (
3718          #1#2
3719          \if_meaning:w - #1 + \else: - \fi:
3720          ( \if_meaning:w - #3 - \fi: #3#4 - \c_one ) / \c_two
3721        )
3722      \fi:
3723      / #3#4
3724    }
```

For the sake of completeness:

```
3725  \cs_new:Npn \int_div_round:nn #1#2
3726    { \__int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }
```

Finally there's the modulus operation.

```
3727  \cs_new:Npn \int_mod:nn #1#2
3728    {
3729      \__int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
3730        \__int_value:w \__int_eval:w #1 \exp_after:wN ;
3731        \__int_value:w \__int_eval:w #2 ;
3732      \__int_eval_end:
3733    }
3734  \cs_new:Npn \__int_mod:ww #1; #2;
3735    { #1 - ( \__int_div_truncate:NwNw #1 ; #2 ; ) * #2 }
```

(*End definition for* `\int_div_truncate:nn` *and others. These functions are documented on page 63.*)

8.2   Creating and initialising integers

`\int_new:N`
`\int_new:c`

Two ways to do this: one for the format and one for the LaTeX 2ε 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.)

```
3736  ⟨*package⟩
3737  \cs_new_protected:Npn \int_new:N #1
3738    {
3739      \__chk_if_free_cs:N #1
3740      \cs:w newcount \cs_end: #1
3741    }
3742  ⟨/package⟩
3743  \cs_generate_variant:Nn \int_new:N { c }
```

(*End definition for* `\int_new:N`. *This function is documented on page 63.*)

`\int_const:Nn`
`\int_const:cn`
`\__int_constdef:Nw`
`\c__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.

```
3744   \cs_new_protected:Npn \int_const:Nn #1#2
3745     {
3746       \int_compare:nNnTF {#2} < \c_zero
3747         {
3748           \int_new:N #1
3749           \int_gset:Nn #1 {#2}
3750         }
3751         {
3752           \int_compare:nNnTF {#2} > \c__max_constdef_int
3753             {
3754               \int_new:N #1
3755               \int_gset:Nn #1 {#2}
3756             }
3757             {
3758               \__chk_if_free_cs:N #1
3759               \tex_global:D \__int_constdef:Nw #1 =
3760                 \__int_eval:w #2 \__int_eval_end:
3761             }
3762         }
3763     }
3764   \cs_generate_variant:Nn \int_const:Nn { c }
3765   \if_int_odd:w 0
3766     \cs_if_exist:NT \luatex_luatexversion:D  { 1 }
3767     \cs_if_exist:NT \uptex_disablecjktoken:D
3768       { \if_int_compare:w \ptex_jis:D "2121 = "3000 ~ 1 \fi: }
3769     \cs_if_exist:NT \xetex_XeTeXversion:D     { 1 } ~
3770       \cs_if_exist:NTF \uptex_disablecjktoken:D
3771         { \cs_new_eq:NN \__int_constdef:Nw \uptex_kchardef:D }
3772         { \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D }
3773       \__int_constdef:Nw \c__max_constdef_int 1114111 ~
3774   \else:
3775     \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
3776     \tex_mathchardef:D \c__max_constdef_int 32767 ~
3777   \fi:
```

(*End definition for* `\int_const:Nn` *,* `\__int_constdef:Nw` *, and* `\c__max_constdef_int` *. These functions are documented on page* 63*.*)

`\int_zero:N`   Functions that reset an ⟨*integer*⟩ register to zero.
`\int_zero:c`
`\int_gzero:N`
`\int_gzero:c`

```
3778   \cs_new_protected:Npn \int_zero:N  #1 { #1 = \c_zero }
3779   \cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero }
3780   \cs_generate_variant:Nn \int_zero:N  { c }
3781   \cs_generate_variant:Nn \int_gzero:N { c }
```

(*End definition for* `\int_zero:N` *and* `\int_gzero:N` *. These functions are documented on page* 63*.*)

`\int_zero_new:N`   Create a register if needed, otherwise clear it.
`\int_zero_new:c`
`\int_gzero_new:N`
`\int_gzero_new:c`

```
3782   \cs_new_protected:Npn \int_zero_new:N  #1
3783     { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }
3784   \cs_new_protected:Npn \int_gzero_new:N #1
3785     { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }
3786   \cs_generate_variant:Nn \int_zero_new:N  { c }
3787   \cs_generate_variant:Nn \int_gzero_new:N { c }
```

(*End definition for* `\int_zero_new:N` *and* `\int_gzero_new:N` *. These functions are documented on page* 63*.*)

`\int_set_eq:NN` Setting equal means using one integer inside the set function of another.

`\int_set_eq:cN`

`\int_set_eq:Nc`
```
3788 \cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }
```
`\int_set_eq:cc`
```
3789 \cs_generate_variant:Nn \int_set_eq:NN {        c }
```
`\int_gset_eq:NN`
```
3790 \cs_generate_variant:Nn \int_set_eq:NN { Nc , cc }
3791 \cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
```
`\int_gset_eq:cN`
```
3792 \cs_generate_variant:Nn \int_gset_eq:NN {        c }
```
`\int_gset_eq:Nc`
```
3793 \cs_generate_variant:Nn \int_gset_eq:NN { Nc , cc }
```
`\int_gset_eq:cc`

(*End definition for* `\int_set_eq:NN` *and* `\int_gset_eq:NN`. *These functions are documented on page* *63*.)

`\int_if_exist_p:N` Copies of the `cs` functions defined in l3basics.

`\int_if_exist_p:c`
```
3794 \prg_new_eq_conditional:NNn \int_if_exist:N \cs_if_exist:N
```
`\int_if_exist:NTF`
```
3795    { TF , T , F , p }
```
`\int_if_exist:cTF`
```
3796 \prg_new_eq_conditional:NNn \int_if_exist:c \cs_if_exist:c
3797    { TF , T , F , p }
```

(*End definition for* `\int_if_exist:NTF`. *This function is documented on page* *64*.)

8.3   Setting and incrementing integers

`\int_add:Nn` Adding and subtracting to and from a counter . . .

`\int_add:cn`
```
3798 \cs_new_protected:Npn \int_add:Nn #1#2
```
`\int_gadd:Nn`
```
3799    { \tex_advance:D #1 by \__int_eval:w #2 \__int_eval_end: }
```
`\int_gadd:cn`
```
3800 \cs_new_protected:Npn \int_sub:Nn #1#2
```
`\int_sub:Nn`
```
3801    { \tex_advance:D #1 by - \__int_eval:w #2 \__int_eval_end: }
```
`\int_sub:cn`
```
3802 \cs_new_protected:Npn \int_gadd:Nn
```
`\int_gsub:Nn`
```
3803    { \tex_global:D \int_add:Nn }
```
`\int_gsub:cn`
```
3804 \cs_new_protected:Npn \int_gsub:Nn
3805    { \tex_global:D \int_sub:Nn }
3806 \cs_generate_variant:Nn \int_add:Nn  { c }
3807 \cs_generate_variant:Nn \int_gadd:Nn { c }
3808 \cs_generate_variant:Nn \int_sub:Nn  { c }
3809 \cs_generate_variant:Nn \int_gsub:Nn { c }
```

(*End definition for* `\int_add:Nn` *and others. These functions are documented on page* *64*.)

`\int_incr:N` Incrementing and decrementing of integer registers is done with the following functions.

`\int_incr:c`
```
3810 \cs_new_protected:Npn \int_incr:N #1
```
`\int_gincr:N`
```
3811    { \tex_advance:D #1 \c_one }
```
`\int_gincr:c`
```
3812 \cs_new_protected:Npn \int_decr:N #1
```
`\int_decr:N`
```
3813    { \tex_advance:D #1 \c_minus_one }
```
`\int_decr:c`
```
3814 \cs_new_protected:Npn \int_gincr:N
```
`\int_gdecr:N`
```
3815    { \tex_global:D \int_incr:N }
```
`\int_gdecr:c`
```
3816 \cs_new_protected:Npn \int_gdecr:N
3817    { \tex_global:D \int_decr:N }
3818 \cs_generate_variant:Nn \int_incr:N  { c }
3819 \cs_generate_variant:Nn \int_decr:N  { c }
3820 \cs_generate_variant:Nn \int_gincr:N { c }
3821 \cs_generate_variant:Nn \int_gdecr:N { c }
```

(*End definition for* `\int_incr:N` *and others. These functions are documented on page* *64*.)

\int_set:Nn
\int_set:cn
\int_gset:Nn
\int_gset:cn

As integers are register-based TeX will issue an error if they are not defined. Thus there is no need for the checking code seen with token list variables.

```
3822 \cs_new_protected:Npn \int_set:Nn #1#2
3823   { #1 ~ \__int_eval:w #2\__int_eval_end: }
3824 \cs_new_protected:Npn \int_gset:Nn { \tex_global:D \int_set:Nn }
3825 \cs_generate_variant:Nn \int_set:Nn  { c }
3826 \cs_generate_variant:Nn \int_gset:Nn { c }
```

(*End definition for* \int_set:Nn *and* \int_gset:Nn. *These functions are documented on page 64.*)

8.4   Using integers

\int_use:N
\int_use:c

Here is how counters are accessed:

```
3827 \cs_new_eq:NN \int_use:N \tex_the:D
```

We hand-code this for some speed gain:

```
3828 %\cs_generate_variant:Nn \int_use:N { c }
3829 \cs_new:Npn \int_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(*End definition for* \int_use:N. *This function is documented on page 64.*)

8.5   Integer expression conditionals

__prg_compare_error:
__prg_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 __prg_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 __prg_compare_error:Nw gets expanded, cleaning up the end of the test and telling the user what the problem was.

```
3830 \cs_new_protected:Npn \__prg_compare_error:
3831   {
3832     \if_int_compare:w \c_zero \c_zero \fi:
3833     =
3834     \__prg_compare_error:
3835   }
3836 \cs_new:Npn \__prg_compare_error:Nw
3837     #1#2 \q_stop
3838   {
3839     { }
3840     \c_zero \fi:
3841     \__msg_kernel_expandable_error:nnn
3842       { kernel } { unknown-comparison } {#1}
3843     \prg_return_false:
3844   }
```

(*End definition for* __prg_compare_error: *and* __prg_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 will terminate it. If the argument contains no relation symbol, __prg_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 will be ended by the two odd-looking items e and {=nd_}, with a trailing \q_stop used to grab the entire argument when necessary.

```
3845 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
3846   {
3847     \exp_after:wN \__int_compare:w
3848     \__int_value:w \__int_eval:w #1 \__prg_compare_error:
3849   }
3850 \cs_new:Npn \__int_compare:w #1 \__prg_compare_error:
3851   {
3852     \exp_after:wN \if_false: \__int_value:w
3853       \__int_compare:Nw #1 e { = nd_ } \q_stop
3854   }
```

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 __prg_compare_error:Nw raises an error.

```
3855 \cs_new:Npn \__int_compare:Nw #1#2 \q_stop
3856   {
3857     \exp_after:wN \__int_compare:NNw
3858       \__int_to_roman:w - 0 #2 \q_mark
3859     #1#2 \q_stop
3860   }
3861 \cs_new:Npn \__int_compare:NNw #1#2#3 \q_mark
3862   {
3863     \etex_unexpanded:D
3864     \use:c
3865       {
3866         __int_compare_ \token_to_str:N #1
3867         \if_meaning:w = #2 = \fi:
```

```
3868            :NNw
3869          }
3870        \__prg_compare_error:Nw #1
3871      }
```

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.

```
3872  \cs_new:cpn { __int_compare_end_=:NNw } #1#2#3 e #4 \q_stop
3873    {
3874      {#3} \exp_stop_f:
3875      \prg_return_false: \else: \prg_return_true: \fi:
3876    }
3877  \cs_new:Npn \__int_compare:nnN #1#2#3
3878    {
3879        {#2} \exp_stop_f:
3880        \prg_return_false: \exp_after:wN \use_none_delimit_by_q_stop:w
3881      \fi:
3882      #1 #2 #3 \exp_after:wN \__int_compare:Nw \__int_value:w \__int_eval:w
3883    }
```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding `\__prg_compare_error:Nw` ⟨*token*⟩ responsible for error detection.

```
3884  \cs_new:cpn { __int_compare_=:NNw } #1#2#3 =
3885    { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
3886  \cs_new:cpn { __int_compare_<:NNw } #1#2#3 <
3887    { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
3888  \cs_new:cpn { __int_compare_>:NNw } #1#2#3 >
3889    { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }
3890  \cs_new:cpn { __int_compare_==:NNw } #1#2#3 ==
3891    { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
3892  \cs_new:cpn { __int_compare_!=:NNw } #1#2#3 !=
3893    { \__int_compare:nnN { \if_int_compare:w } {#3} = }
3894  \cs_new:cpn { __int_compare_<=:NNw } #1#2#3 <=
3895    { \__int_compare:nnN { \if_int_compare:w } {#3} > }
3896  \cs_new:cpn { __int_compare_>=:NNw } #1#2#3 >=
3897    { \__int_compare:nnN { \if_int_compare:w } {#3} < }
```

(*End definition for* `\int_compare:nTF` *and others. These functions are documented on page 65.*)

`\int_compare_p:nNn`
`\int_compare:nNnTF`   More efficient but less natural in typing.

```
3898  \prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
3899    {
3900      \if_int_compare:w \__int_eval:w #1 #2 \__int_eval:w #3 \__int_eval_end:
3901        \prg_return_true:
3902      \else:
```

```
3903        \prg_return_false:
3904      \fi:
3905    }
```

(End definition for `\int_compare:nNnTF`*. This function is documented on page 65.)*

`\int_case:nn` For integer cases, the first task to fully expand the check condition. The over all idea is
`\int_case:nnTF` then much the same as for `\str_case:nn(TF)` as described in l3basics.
`\__int_case:nnTF`
`\__int_case:nw`
`\__int_case_end:nw`

```
3906 \cs_new:Npn \int_case:nnTF #1
3907    {
3908      \exp:w
3909      \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
3910    }
3911 \cs_new:Npn \int_case:nnT #1#2#3
3912    {
3913      \exp:w
3914      \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
3915    }
3916 \cs_new:Npn \int_case:nnF #1#2
3917    {
3918      \exp:w
3919      \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
3920    }
3921 \cs_new:Npn \int_case:nn #1#2
3922    {
3923      \exp:w
3924      \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
3925    }
3926 \cs_new:Npn \__int_case:nnTF #1#2#3#4
3927    { \__int_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
3928 \cs_new:Npn \__int_case:nw #1#2#3
3929    {
3930      \int_compare:nNnTF {#1} = {#2}
3931        { \__int_case_end:nw {#3} }
3932        { \__int_case:nw {#1} }
3933    }
3934 \cs_new_eq:NN \__int_case_end:nw \__prg_case_end:nw
```

(End definition for `\int_case:nnTF` *and others. These functions are documented on page 66.)*

`\int_if_odd_p:n` A predicate function.
`\int_if_odd:nTF`
`\int_if_even_p:n`
`\int_if_even:nTF`

```
3935 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
3936    {
3937      \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
3938        \prg_return_true:
3939      \else:
3940        \prg_return_false:
3941      \fi:
3942    }
3943 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
3944    {
3945      \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
3946        \prg_return_false:
3947      \else:
```

```
3948          \prg_return_true:
3949        \fi:
3950      }
```

(*End definition for* `\int_if_odd:nTF` *and* `\int_if_even:nTF`. *These functions are documented on page* *66*.)

8.6   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.

```
3951 \cs_new:Npn \int_while_do:nn #1#2
3952   {
3953      \int_compare:nT {#1}
3954        {
3955          #2
3956          \int_while_do:nn {#1} {#2}
3957        }
3958   }
3959 \cs_new:Npn \int_until_do:nn #1#2
3960   {
3961      \int_compare:nF {#1}
3962        {
3963          #2
3964          \int_until_do:nn {#1} {#2}
3965        }
3966   }
3967 \cs_new:Npn \int_do_while:nn #1#2
3968   {
3969      #2
3970      \int_compare:nT {#1}
3971        { \int_do_while:nn {#1} {#2} }
3972   }
3973 \cs_new:Npn \int_do_until:nn #1#2
3974   {
3975      #2
3976      \int_compare:nF {#1}
3977        { \int_do_until:nn {#1} {#2} }
3978   }
```

(*End definition for* `\int_while_do:nn` *and others. These functions are documented on page* *67*.)

`\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.

```
3979 \cs_new:Npn \int_while_do:nNnn #1#2#3#4
3980   {
3981      \int_compare:nNnT {#1} #2 {#3}
3982        {
3983          #4
3984          \int_while_do:nNnn {#1} #2 {#3} {#4}
3985        }
3986   }
3987 \cs_new:Npn \int_until_do:nNnn #1#2#3#4
3988   {
3989      \int_compare:nNnF {#1} #2 {#3}
```

337

```
3990          {
3991            #4
3992            \int_until_do:nNnn {#1} #2 {#3} {#4}
3993          }
3994      }
3995  \cs_new:Npn \int_do_while:nNnn #1#2#3#4
3996      {
3997        #4
3998        \int_compare:nNnT {#1} #2 {#3}
3999          { \int_do_while:nNnn {#1} #2 {#3} {#4} }
4000      }
4001  \cs_new:Npn \int_do_until:nNnn #1#2#3#4
4002      {
4003        #4
4004        \int_compare:nNnF {#1} #2 {#3}
4005          { \int_do_until:nNnn {#1} #2 {#3} {#4} }
4006      }
```

(*End definition for* `\int_while_do:nNnn` *and others. These functions are documented on page 67.*)

8.7 Integer step functions

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).

```
4007  \cs_new:Npn \int_step_function:nnnN #1#2#3
4008      {
4009        \exp_after:wN \__int_step:wwwN
4010        \__int_value:w \__int_eval:w #1 \exp_after:wN ;
4011        \__int_value:w \__int_eval:w #2 \exp_after:wN ;
4012        \__int_value:w \__int_eval:w #3 ;
4013      }
4014  \cs_new:Npn \__int_step:wwwN #1; #2; #3; #4
4015      {
4016        \int_compare:nNnTF {#2} > \c_zero
4017          { \__int_step:NnnnN > }
4018          {
4019            \int_compare:nNnTF {#2} = \c_zero
4020              {
4021                \__msg_kernel_expandable_error:nnn { kernel } { zero-step } {#4}
4022                \use_none:nnnn
4023              }
4024              { \__int_step:NnnnN < }
4025          }
4026        {#1} {#2} {#3} #4
4027      }
4028  \cs_new:Npn \__int_step:NnnnN #1#2#3#4#5
4029      {
4030        \int_compare:nNnF {#2} #1 {#4}
4031          {
4032            #5 {#2}
4033            \exp_args:NNf \__int_step:NnnnN
```

```
4034                #1 { \int_eval:n { #2 + #3 } } } {#3} {#4} #5
4035            }
4036        }
```

(*End definition for* `\int_step_function:nnnN` *,* `\__int_step:wwwN` *, and* `\__int_step:NnnnN`*. These functions are documented on page* *68*.)

<code>\int_step_inline:nnnn</code>
<code>\int_step_variable:nnnNn</code>
<code>__int_step:NNnnnn</code>

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__prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so no breaking function will recognize this break point as its own.

```
4037 \cs_new_protected:Npn \int_step_inline:nnnn
4038    {
4039        \int_gincr:N \g__prg_map_int
4040        \exp_args:NNc \__int_step:NNnnnn
4041            \cs_gset:Npn
4042            { __prg_map_ \int_use:N \g__prg_map_int :w }
4043    }
4044 \cs_new_protected:Npn \int_step_variable:nnnNn #1#2#3#4#5
4045    {
4046        \int_gincr:N \g__prg_map_int
4047        \exp_args:NNc \__int_step:NNnnnn
4048            \cs_gset:Npx
4049            { __prg_map_ \int_use:N \g__prg_map_int :w }
4050            {#1}{#2}{#3}
4051            {
4052                \tl_set:Nn \exp_not:N #4 {##1}
4053                \exp_not:n {#5}
4054            }
4055    }
4056 \cs_new_protected:Npn \__int_step:NNnnnn #1#2#3#4#5#6
4057    {
4058        #1 #2 ##1 {#6}
4059        \int_step_function:nnnN {#3} {#4} {#5} #2
4060        \__prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__prg_map_int }
4061    }
```

(*End definition for* `\int_step_inline:nnnn` *,* `\int_step_variable:nnnNn` *, and* `\__int_step:NNnnnn`*. These functions are documented on page* *68*.)

8.8 Formatting integers

<code>\int_to_arabic:n</code> Nothing exciting here.

```
4062 \cs_new_eq:NN \int_to_arabic:n \int_eval:n
```

(*End definition for* `\int_to_arabic:n`*. This function is documented on page* *68*.)

<code>\int_to_symbols:nnn</code>
<code>__int_to_symbols:nnnn</code>

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

339

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.

```
4063 \cs_new:Npn \int_to_symbols:nnn #1#2#3
4064   {
4065     \int_compare:nNnTF {#1} > {#2}
4066       {
4067         \exp_args:NNo \exp_args:No \__int_to_symbols:nnnn
4068           {
4069             \int_case:nn
4070               { 1 + \int_mod:nn { #1 - 1 } {#2} }
4071               {#3}
4072           }
4073           {#1} {#2} {#3}
4074       }
4075       { \int_case:nn {#1} {#3} }
4076   }
4077 \cs_new:Npn \__int_to_symbols:nnnn #1#2#3#4
4078   {
4079     \exp_args:Nf \int_to_symbols:nnn
4080       { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}
4081       #1
4082   }
```

(*End definition for* \int_to_symbols:nnn *and* __int_to_symbols:nnnn. *These functions are documented on page 69.*)

\int_to_alph:n   These both use the above function with input functions that make sense for the alphabet
\int_to_Alph:n   in English.

```
4083 \cs_new:Npn \int_to_alph:n #1
4084   {
4085     \int_to_symbols:nnn {#1} { 26 }
4086       {
4087         {  1 } { a }
4088         {  2 } { b }
4089         {  3 } { c }
4090         {  4 } { d }
4091         {  5 } { e }
4092         {  6 } { f }
4093         {  7 } { g }
4094         {  8 } { h }
4095         {  9 } { i }
4096         { 10 } { j }
4097         { 11 } { k }
4098         { 12 } { l }
4099         { 13 } { m }
4100         { 14 } { n }
4101         { 15 } { o }
4102         { 16 } { p }
4103         { 17 } { q }
4104         { 18 } { r }
4105         { 19 } { s }
4106         { 20 } { t }
4107         { 21 } { u }
4108         { 22 } { v }
```

340

```
4109        { 23 } { w }
4110        { 24 } { x }
4111        { 25 } { y }
4112        { 26 } { z }
4113      }
4114   }
4115 \cs_new:Npn \int_to_Alph:n #1
4116   {
4117     \int_to_symbols:nnn {#1} { 26 }
4118       {
4119         {  1 } { A }
4120         {  2 } { B }
4121         {  3 } { C }
4122         {  4 } { D }
4123         {  5 } { E }
4124         {  6 } { F }
4125         {  7 } { G }
4126         {  8 } { H }
4127         {  9 } { I }
4128         { 10 } { J }
4129         { 11 } { K }
4130         { 12 } { L }
4131         { 13 } { M }
4132         { 14 } { N }
4133         { 15 } { O }
4134         { 16 } { P }
4135         { 17 } { Q }
4136         { 18 } { R }
4137         { 19 } { S }
4138         { 20 } { T }
4139         { 21 } { U }
4140         { 22 } { V }
4141         { 23 } { W }
4142         { 24 } { X }
4143         { 25 } { Y }
4144         { 26 } { Z }
4145       }
4146   }
```

(End definition for `\int_to_alph:n` *and* `\int_to_Alph:n`. *These functions are documented on page 69.)*

`\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.

```
4147 \cs_new:Npn \int_to_base:nn #1
4148   { \exp_args:Nf \__int_to_base:nn { \int_eval:n {#1} } }
4149 \cs_new:Npn \int_to_Base:nn #1
4150   { \exp_args:Nf \__int_to_Base:nn { \int_eval:n {#1} } }
4151 \cs_new:Npn \__int_to_base:nn #1#2
4152   {
4153     \int_compare:nNnTF {#1} < \c_zero
4154       { \exp_args:No \__int_to_base:nnN { \use_none:n #1 } {#2} - }
4155       { \__int_to_base:nnN {#1} {#2} \c_empty_tl }
4156   }
```

341

```
4157 \cs_new:Npn \__int_to_Base:nn #1#2
4158   {
4159     \int_compare:nNnTF {#1} < \c_zero
4160       { \exp_args:No \__int_to_Base:nnN { \use_none:n #1 } {#2} - }
4161       { \__int_to_Base:nnN {#1} {#2} \c_empty_tl }
4162   }
```

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.

```
4163 \cs_new:Npn \__int_to_base:nnN #1#2#3
4164   {
4165     \int_compare:nNnTF {#1} < {#2}
4166       { \exp_last_unbraced:Nf #3 { \__int_to_letter:n {#1} } }
4167       {
4168         \exp_args:Nf \__int_to_base:nnnN
4169           { \__int_to_letter:n { \int_mod:nn {#1} {#2} } }
4170           {#1}
4171           {#2}
4172           #3
4173       }
4174   }
4175 \cs_new:Npn \__int_to_base:nnnN #1#2#3#4
4176   {
4177     \exp_args:Nf \__int_to_base:nnN
4178       { \int_div_truncate:nn {#2} {#3} }
4179       {#3}
4180       #4
4181       #1
4182   }
4183 \cs_new:Npn \__int_to_Base:nnN #1#2#3
4184   {
4185     \int_compare:nNnTF {#1} < {#2}
4186       { \exp_last_unbraced:Nf #3 { \__int_to_Letter:n {#1} } }
4187       {
4188         \exp_args:Nf \__int_to_Base:nnnN
4189           { \__int_to_Letter:n { \int_mod:nn {#1} {#2} } }
4190           {#1}
4191           {#2}
4192           #3
4193       }
4194   }
4195 \cs_new:Npn \__int_to_Base:nnnN #1#2#3#4
4196   {
4197     \exp_args:Nf \__int_to_Base:nnN
4198       { \int_div_truncate:nn {#2} {#3} }
4199       {#3}
4200       #4
4201       #1
4202   }
```

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:`.

```
4203 \cs_new:Npn \__int_to_letter:n #1
4204   {
4205     \exp_after:wN \exp_after:wN
4206     \if_case:w \__int_eval:w #1 - \c_ten \__int_eval_end:
4207         a
4208     \or: b
4209     \or: c
4210     \or: d
4211     \or: e
4212     \or: f
4213     \or: g
4214     \or: h
4215     \or: i
4216     \or: j
4217     \or: k
4218     \or: l
4219     \or: m
4220     \or: n
4221     \or: o
4222     \or: p
4223     \or: q
4224     \or: r
4225     \or: s
4226     \or: t
4227     \or: u
4228     \or: v
4229     \or: w
4230     \or: x
4231     \or: y
4232     \or: z
4233     \else: \__int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
4234     \fi:
4235   }
4236 \cs_new:Npn \__int_to_Letter:n #1
4237   {
4238     \exp_after:wN \exp_after:wN
4239     \if_case:w \__int_eval:w #1 - \c_ten \__int_eval_end:
4240         A
4241     \or: B
4242     \or: C
4243     \or: D
4244     \or: E
4245     \or: F
4246     \or: G
4247     \or: H
4248     \or: I
4249     \or: J
4250     \or: K
```

```
4251        \or: L
4252        \or: M
4253        \or: N
4254        \or: O
4255        \or: P
4256        \or: Q
4257        \or: R
4258        \or: S
4259        \or: T
4260        \or: U
4261        \or: V
4262        \or: W
4263        \or: X
4264        \or: Y
4265        \or: Z
4266        \else: \__int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
4267        \fi:
4268      }
```

(*End definition for* \int_to_base:nn *and others. These functions are documented on page 70.*)

\int_to_bin:n   Wrappers around the generic function.
\int_to_hex:n
\int_to_Hex:n
\int_to_oct:n
```
4269  \cs_new:Npn \int_to_bin:n #1
4270    { \int_to_base:nn {#1} { 2 } }
4271  \cs_new:Npn \int_to_hex:n #1
4272    { \int_to_base:nn {#1} { 16 } }
4273  \cs_new:Npn \int_to_Hex:n #1
4274    { \int_to_Base:nn {#1} { 16 } }
4275  \cs_new:Npn \int_to_oct:n #1
4276    { \int_to_base:nn {#1} { 8 } }
```

(*End definition for* \int_to_bin:n *and others. These functions are documented on page 69.*)

\int_to_roman:n   The __int_to_roman:w primitive creates tokens of category code 12 (other). Usually,
\int_to_Roman:n   what is actually wanted is letters. The approach here is to convert the output of the
__int_to_roman:N   primitive into letters using appropriate control sequence names. That keeps everything
__int_to_roman:N   expandable. The loop will be terminated by the conversion of the Q.
__int_to_roman_i:w
__int_to_roman_v:w
```
4277  \cs_new:Npn \int_to_roman:n #1
4278    {
4279      \exp_after:wN \__int_to_roman:N
4280        \__int_to_roman:w \int_eval:n {#1} Q
4281    }
4282  \cs_new:Npn \__int_to_roman:N #1
4283    {
4284      \use:c { __int_to_roman_ #1 :w }
4285      \__int_to_roman:N
4286    }
4287  \cs_new:Npn \int_to_Roman:n #1
4288    {
4289      \exp_after:wN \__int_to_Roman_aux:N
4290        \__int_to_roman:w \int_eval:n {#1} Q
4291    }
4292  \cs_new:Npn \__int_to_Roman_aux:N #1
4293    {
```
__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

```
4294        \use:c { __int_to_Roman_ #1 :w }
4295        \__int_to_Roman_aux:N
4296      }
4297  \cs_new:Npn \__int_to_roman_i:w { i }
4298  \cs_new:Npn \__int_to_roman_v:w { v }
4299  \cs_new:Npn \__int_to_roman_x:w { x }
4300  \cs_new:Npn \__int_to_roman_l:w { l }
4301  \cs_new:Npn \__int_to_roman_c:w { c }
4302  \cs_new:Npn \__int_to_roman_d:w { d }
4303  \cs_new:Npn \__int_to_roman_m:w { m }
4304  \cs_new:Npn \__int_to_roman_Q:w #1 { }
4305  \cs_new:Npn \__int_to_Roman_i:w { I }
4306  \cs_new:Npn \__int_to_Roman_v:w { V }
4307  \cs_new:Npn \__int_to_Roman_x:w { X }
4308  \cs_new:Npn \__int_to_Roman_l:w { L }
4309  \cs_new:Npn \__int_to_Roman_c:w { C }
4310  \cs_new:Npn \__int_to_Roman_d:w { D }
4311  \cs_new:Npn \__int_to_Roman_m:w { M }
4312  \cs_new:Npn \__int_to_Roman_Q:w #1 { }
```

(*End definition for* \int_to_roman:n *and others. These functions are documented on page 70.*)

8.9   Converting from other formats to integers

__int_pass_signs:wn
__int_pass_signs_end:wn

Called as __int_pass_signs:wn ⟨*signs and digits*⟩ \q_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 \q_stop). More precisely, it deletes any + and passes any - to the input stream, hence should be called in an integer expression.

```
4313  \cs_new:Npn \__int_pass_signs:wn #1
4314    {
4315      \if:w + \if:w - \exp_not:N #1 + \fi: \exp_not:N #1
4316        \exp_after:wN \__int_pass_signs:wn
4317      \else:
4318        \exp_after:wN \__int_pass_signs_end:wn
4319        \exp_after:wN #1
4320      \fi:
4321    }
4322  \cs_new:Npn \__int_pass_signs_end:wn #1 \q_stop #2 { #2 #1 }
```

(*End 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.

```
4323  \cs_new:Npn \int_from_alph:n #1
4324    {
4325      \int_eval:n
4326        {
4327          \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
4328            \q_stop { \__int_from_alph:nN { 0 } }
4329          \q_recursion_tail \q_recursion_stop
4330        }
4331    }
```

```
4332 \cs_new:Npn \__int_from_alph:nN #1#2
4333   {
4334     \quark_if_recursion_tail_stop_do:Nn #2 {#1}
4335     \exp_args:Nf \__int_from_alph:nN
4336       { \int_eval:n { #1 * 26 + \__int_from_alph:N #2 } }
4337   }
4338 \cs_new:Npn \__int_from_alph:N #1
4339   { `#1 - \int_compare:nNnTF { `#1 } < { 91 } { 64 } { 96 } }
```

(*End definition for* `\int_from_alph:n`*,* `\__int_from_alph:nN`*, and* `\__int_from_alph:N`*. These functions are documented on page 70.*)

`\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.

```
4340 \cs_new:Npn \int_from_base:nn #1#2
4341   {
4342     \int_eval:n
4343       {
4344         \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
4345           \q_stop { \__int_from_base:nnN { 0 } {#2} }
4346         \q_recursion_tail \q_recursion_stop
4347       }
4348   }
4349 \cs_new:Npn \__int_from_base:nnN #1#2#3
4350   {
4351     \quark_if_recursion_tail_stop_do:Nn #3 {#1}
4352     \exp_args:Nf \__int_from_base:nnN
4353       { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
4354       {#2}
4355   }
4356 \cs_new:Npn \__int_from_base:N #1
4357   {
4358     \int_compare:nNnTF { `#1 } < { 58 }
4359       {#1}
4360       { `#1 - \int_compare:nNnTF { `#1 } < { 91 } { 55 } { 87 } }
4361   }
```

(*End definition for* `\int_from_base:nn`*,* `\__int_from_base:nnN`*, and* `\__int_from_base:N`*. These functions are documented on page 71.*)

`\int_from_bin:n`
`\int_from_hex:n`
`\int_from_oct:n`

Wrappers around the generic function.

```
4362 \cs_new:Npn \int_from_bin:n #1
4363   { \int_from_base:nn {#1} \c_two }
4364 \cs_new:Npn \int_from_hex:n #1
4365   { \int_from_base:nn {#1} \c_sixteen }
4366 \cs_new:Npn \int_from_oct:n #1
4367   { \int_from_base:nn {#1} \c_eight }
```

(*End definition for* `\int_from_bin:n`*,* `\int_from_hex:n`*, and* `\int_from_oct:n`*. These functions are documented on page 70.*)

346

`\c__int_from_roman_i_int`	Constants used to convert from Roman numerals to integers.
`\c__int_from_roman_v_int`	`4368 \int_const:cn { c__int_from_roman_i_int } { 1 }`
`\c__int_from_roman_x_int`	`4369 \int_const:cn { c__int_from_roman_v_int } { 5 }`
`\c__int_from_roman_l_int`	`4370 \int_const:cn { c__int_from_roman_x_int } { 10 }`
`\c__int_from_roman_c_int`	`4371 \int_const:cn { c__int_from_roman_l_int } { 50 }`
`\c__int_from_roman_d_int`	`4372 \int_const:cn { c__int_from_roman_c_int } { 100 }`
`\c__int_from_roman_m_int`	`4373 \int_const:cn { c__int_from_roman_d_int } { 500 }`
`\c__int_from_roman_I_int`	`4374 \int_const:cn { c__int_from_roman_m_int } { 1000 }`
`\c__int_from_roman_V_int`	`4375 \int_const:cn { c__int_from_roman_I_int } { 1 }`
`\c__int_from_roman_X_int`	`4376 \int_const:cn { c__int_from_roman_V_int } { 5 }`
`\c__int_from_roman_L_int`	`4377 \int_const:cn { c__int_from_roman_X_int } { 10 }`
`\c__int_from_roman_C_int`	`4378 \int_const:cn { c__int_from_roman_L_int } { 50 }`
`\c__int_from_roman_D_int`	`4379 \int_const:cn { c__int_from_roman_C_int } { 100 }`
`\c__int_from_roman_M_int`	`4380 \int_const:cn { c__int_from_roman_D_int } { 500 }`
	`4381 \int_const:cn { c__int_from_roman_M_int } { 1000 }`

(*End 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.

```
4382 \cs_new:Npn \int_from_roman:n #1
4383   {
4384     \int_eval:n
4385       {
4386         (
4387           \c_zero
4388           \exp_after:wN \__int_from_roman:NN \tl_to_str:n {#1}
4389           \q_recursion_tail \q_recursion_tail \q_recursion_stop
4390         )
4391       }
4392   }
4393 \cs_new:Npn \__int_from_roman:NN #1#2
4394   {
4395     \quark_if_recursion_tail_stop:N #1
4396     \int_if_exist:cF { c__int_from_roman_ #1 _int }
4397       { \__int_from_roman_error:w }
4398     \quark_if_recursion_tail_stop_do:Nn #2
4399       { + \use:c { c__int_from_roman_ #1 _int } }
4400     \int_if_exist:cF { c__int_from_roman_ #2 _int }
4401       { \__int_from_roman_error:w }
4402     \int_compare:nNnTF
4403       { \use:c { c__int_from_roman_ #1 _int } }
4404       <
4405       { \use:c { c__int_from_roman_ #2 _int } }
4406       {
4407         + \use:c { c__int_from_roman_ #2 _int }
4408         - \use:c { c__int_from_roman_ #1 _int }
4409         \__int_from_roman:NN
4410       }
4411       {
4412         + \use:c { c__int_from_roman_ #1 _int }
4413         \__int_from_roman:NN #2
```

```
4414          }
4415      }
4416  \cs_new:Npn \__int_from_roman_error:w #1 \q_recursion_stop #2
4417      { #2 * \c_zero - \c_one }
```

(*End definition for* \int_from_roman:n, __int_from_roman:NN, *and* __int_from_roman_error:w. *These functions are documented on page 71.*)

8.10   Viewing integer

\int_show:N
\int_show:c
This is very similar to other registers done using __kernel_register_show:N, but differs because the variable #1 may be \currentgrouplevel or \currentgrouptype, in which case the value must be expanded in the current scope rather than when processing \iow_wrap:nnnN.

```
4418  \cs_new_protected:Npn \int_show:N #1
4419      {
4420        \use:x
4421          {
4422            \exp_not:n
4423              { \__msg_show_variable:NNNnn #1 \cs_if_exist:NTF ? { } }
4424                  { > ~ \token_to_str:N #1 = \tex_the:D #1 }
4425          }
4426      }
4427  \cs_generate_variant:Nn \int_show:N { c }
```

(*End definition for* \int_show:N. *This function is documented on page 71.*)

\int_show:n We don't use the TeX primitive \showthe to show integer expressions: this gives a more unified output.

```
4428  \cs_new_protected:Npn \int_show:n
4429      { \__msg_show_wrap:Nn \int_eval:n }
```

(*End definition for* \int_show:n. *This function is documented on page 71.*)

8.11   Constant integers

\c_minus_one This is needed early, and so is in l3basics

(*End definition for* \c_minus_one. *This variable is documented on page 72.*)

\c_zero Again, in l3basics
\c_sixteen

(*End definition for* \c_zero *and* \c_sixteen. *These variables are documented on page 72.*)

\c_one Low-number values not previously defined.
\c_two
\c_three
\c_four
\c_five
\c_six
\c_seven
\c_eight
\c_nine
\c_ten
\c_eleven
\c_twelve
\c_thirteen
\c_fourteen
\c_fifteen

```
4430  \int_const:Nn \c_one      {  1 }
4431  \int_const:Nn \c_two      {  2 }
4432  \int_const:Nn \c_three    {  3 }
4433  \int_const:Nn \c_four     {  4 }
4434  \int_const:Nn \c_five     {  5 }
4435  \int_const:Nn \c_six      {  6 }
4436  \int_const:Nn \c_seven    {  7 }
4437  \int_const:Nn \c_eight    {  8 }
4438  \int_const:Nn \c_nine     {  9 }
4439  \int_const:Nn \c_ten      { 10 }
```

348

```
4440 \int_const:Nn \c_eleven   { 11 }
4441 \int_const:Nn \c_twelve   { 12 }
4442 \int_const:Nn \c_thirteen { 13 }
4443 \int_const:Nn \c_fourteen { 14 }
4444 \int_const:Nn \c_fifteen  { 15 }
```

(*End definition for* \c_one *and others. These variables are documented on page 72.*)

\c_thirty_two   One middling value.

```
4445 \int_const:Nn \c_thirty_two { 32 }
```

(*End definition for* \c_thirty_two. *This variable is documented on page 72.*)

\c_two_hundred_fifty_five
\c_two_hundred_fifty_six   Two classic mid-range integer constants.

```
4446 \int_const:Nn \c_two_hundred_fifty_five { 255 }
4447 \int_const:Nn \c_two_hundred_fifty_six  { 256 }
```

(*End definition for* \c_two_hundred_fifty_five *and* \c_two_hundred_fifty_six. *These variables are documented on page 72.*)

\c_one_hundred
\c_one_thousand
\c_ten_thousand   Simple runs of powers of ten.

```
4448 \int_const:Nn \c_one_hundred  {   100 }
4449 \int_const:Nn \c_one_thousand {  1000 }
4450 \int_const:Nn \c_ten_thousand { 10000 }
```

(*End definition for* \c_one_hundred, \c_one_thousand, *and* \c_ten_thousand. *These variables are documented on page 72.*)

\c_max_int   The largest number allowed is $2^{31} - 1$

```
4451 \int_const:Nn \c_max_int { 2 147 483 647 }
```

(*End definition for* \c_max_int. *This variable is documented on page 72.*)

\c_max_char_int   The largest character code is 1114111 (hexadecimal 10FFFF) in X∃TEX 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 $[0, 255]$.

```
4452 \int_const:Nn \c_max_char_int
4453   {
4454     \if_int_odd:w 0
4455       \cs_if_exist:NT \luatex_luatexversion:D  { 1 }
4456       \cs_if_exist:NT \xetex_XeTeXversion:D    { 1 } ~
4457       "10FFFF
4458     \else:
4459       "FF
4460     \fi:
4461   }
```

(*End definition for* \c_max_char_int. *This variable is documented on page 72.*)

8.12 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.

```
4462 \int_new:N \l_tmpa_int
4463 \int_new:N \l_tmpb_int
4464 \int_new:N \g_tmpa_int
4465 \int_new:N \g_tmpb_int
```

(*End definition for* \l_tmpa_int *and others. These variables are documented on page 72.*)

```
4466 ⟨/initex | package⟩
```

9   l3skip implementation

```
4467 ⟨*initex | package⟩
```

```
4468 ⟨@@=dim⟩
```

9.1 Length primitives renamed

\if_dim:w
__dim_eval:w
__dim_eval_end:

Primitives renamed.

```
4469 \cs_new_eq:NN \if_dim:w        \tex_ifdim:D
4470 \cs_new_eq:NN \__dim_eval:w      \etex_dimexpr:D
4471 \cs_new_eq:NN \__dim_eval_end:   \tex_relax:D
```

(*End definition for* \if_dim:w, __dim_eval:w, *and* __dim_eval_end:. *These functions are documented on page 88.*)

9.2 Creating and initialising dim variables

\dim_new:N
\dim_new:c

Allocating ⟨*dim*⟩ registers . . .

```
4472 ⟨*package⟩
4473 \cs_new_protected:Npn \dim_new:N #1
4474   {
4475     \__chk_if_free_cs:N #1
4476     \cs:w newdimen \cs_end: #1
4477   }
4478 ⟨/package⟩
4479 \cs_generate_variant:Nn \dim_new:N { c }
```

(*End definition for* \dim_new:N. *This function is documented on page 75.*)

\dim_const:Nn
\dim_const:cn

Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
4480 \cs_new_protected:Npn \dim_const:Nn #1
4481   {
4482     \dim_new:N #1
4483     \dim_gset:Nn #1
4484   }
4485 \cs_generate_variant:Nn \dim_const:Nn { c }
```

(*End definition for* \dim_const:Nn. *This function is documented on page 75.*)

\dim_zero:N
\dim_zero:c
\dim_gzero:N
\dim_gzero:c

Reset the register to zero.

```
4486 \cs_new_protected:Npn \dim_zero:N #1 { #1 \c_zero_dim }
4487 \cs_new_protected:Npn \dim_gzero:N { \tex_global:D \dim_zero:N }
4488 \cs_generate_variant:Nn \dim_zero:N  { c }
4489 \cs_generate_variant:Nn \dim_gzero:N { c }
```

(*End definition for* `\dim_zero:N` *and* `\dim_gzero:N`. *These functions are documented on page 75.*)

`\dim_zero_new:N` Create a register if needed, otherwise clear it.
`\dim_zero_new:c`
`\dim_gzero_new:N`
`\dim_gzero_new:c`

```
4490 \cs_new_protected:Npn \dim_zero_new:N #1
4491   { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
4492 \cs_new_protected:Npn \dim_gzero_new:N #1
4493   { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
4494 \cs_generate_variant:Nn \dim_zero_new:N  { c }
4495 \cs_generate_variant:Nn \dim_gzero_new:N { c }
```

(*End definition for* `\dim_zero_new:N` *and* `\dim_gzero_new:N`. *These functions are documented on page 75.*)

`\dim_if_exist_p:N` Copies of the `cs` functions defined in l3basics.
`\dim_if_exist_p:c`
`\dim_if_exist:N`*TF*
`\dim_if_exist:c`*TF*

```
4496 \prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N
4497   { TF , T , F , p }
4498 \prg_new_eq_conditional:NNn \dim_if_exist:c \cs_if_exist:c
4499   { TF , T , F , p }
```

(*End definition for* `\dim_if_exist:NTF`. *This function is documented on page 75.*)

9.3 Setting dim variables

`\dim_set:Nn` Setting dimensions is easy enough.
`\dim_set:cn`
`\dim_gset:Nn`
`\dim_gset:cn`

```
4500 \cs_new_protected:Npn \dim_set:Nn #1#2
4501   { #1 ~ \__dim_eval:w #2 \__dim_eval_end: }
4502 \cs_new_protected:Npn \dim_gset:Nn { \tex_global:D \dim_set:Nn }
4503 \cs_generate_variant:Nn \dim_set:Nn  { c }
4504 \cs_generate_variant:Nn \dim_gset:Nn { c }
```

(*End definition for* `\dim_set:Nn` *and* `\dim_gset:Nn`. *These functions are documented on page 76.*)

`\dim_set_eq:NN` All straightforward.
`\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`

```
4505 \cs_new_protected:Npn \dim_set_eq:NN #1#2 { #1 = #2 }
4506 \cs_generate_variant:Nn \dim_set_eq:NN {       c }
4507 \cs_generate_variant:Nn \dim_set_eq:NN { Nc , cc }
4508 \cs_new_protected:Npn \dim_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
4509 \cs_generate_variant:Nn \dim_gset_eq:NN {       c }
4510 \cs_generate_variant:Nn \dim_gset_eq:NN { Nc , cc }
```

(*End definition for* `\dim_set_eq:NN` *and* `\dim_gset_eq:NN`. *These functions are documented on page 76.*)

`\dim_add:Nn` Using `by` here deals with the (incorrect) case `\dimen123`.
`\dim_add:cn`
`\dim_gadd:Nn`
`\dim_gadd:cn`
`\dim_sub:Nn`
`\dim_sub:cn`
`\dim_gsub:Nn`
`\dim_gsub:cn`

```
4511 \cs_new_protected:Npn \dim_add:Nn #1#2
4512   { \tex_advance:D #1 by \__dim_eval:w #2 \__dim_eval_end: }
4513 \cs_new_protected:Npn \dim_gadd:Nn { \tex_global:D \dim_add:Nn }
4514 \cs_generate_variant:Nn \dim_add:Nn  { c }
4515 \cs_generate_variant:Nn \dim_gadd:Nn { c }
4516 \cs_new_protected:Npn \dim_sub:Nn #1#2
4517   { \tex_advance:D #1 by - \__dim_eval:w #2 \__dim_eval_end: }
4518 \cs_new_protected:Npn \dim_gsub:Nn { \tex_global:D \dim_sub:Nn }
4519 \cs_generate_variant:Nn \dim_sub:Nn  { c }
4520 \cs_generate_variant:Nn \dim_gsub:Nn { c }
```

(*End definition for* `\dim_add:Nn` *and others. These functions are documented on page 76.*)

351

9.4 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.

```
4521 \cs_new:Npn \dim_abs:n #1
4522   {
4523     \exp_after:wN \__dim_abs:N
4524     \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
4525   }
4526 \cs_new:Npn \__dim_abs:N #1
4527   { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
4528 \cs_set:Npn \dim_max:nn #1#2
4529   {
4530     \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
4531       \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
4532       \dim_use:N \__dim_eval:w #2 ;
4533       >
4534     \__dim_eval_end:
4535   }
4536 \cs_set:Npn \dim_min:nn #1#2
4537   {
4538     \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
4539       \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
4540       \dim_use:N \__dim_eval:w #2 ;
4541       <
4542     \__dim_eval_end:
4543   }
4544 \cs_new:Npn \__dim_maxmin:wwN #1 ; #2 ; #3
4545   {
4546     \if_dim:w #1 #3 #2 ~
4547       #1
4548     \else:
4549       #2
4550     \fi:
4551   }
```

(*End definition for* `\dim_abs:n` *and others. These functions are documented on page 76.*)

`\dim_ratio:nn`
`\__dim_ratio:n`

With dimension expressions, something like 10 pt * (5 pt / 10 pt) will 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.

```
4552 \cs_new:Npn \dim_ratio:nn #1#2
4553   { \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
4554 \cs_new:Npn \__dim_ratio:n #1
4555   { \__int_value:w \__dim_eval:w #1 \__dim_eval_end: }
```

(*End definition for* `\dim_ratio:nn` *and* `\__dim_ratio:n`. *These functions are documented on page 77.*)

9.5 Dimension expression conditionals

`\dim_compare_p:nNn`
`\dim_compare:nNn`*TF*

Simple comparison.

```
4556 \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
4557   {
4558     \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
```

```
4559            \prg_return_true: \else: \prg_return_false: \fi:
4560        }
```

(*End definition for* `\dim_compare:nNnTF`*. This function is documented on page [77](#).*)

`\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`

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 `\__prg_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 `>`.

```
4561 \prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
4562    {
4563        \exp_after:wN \__dim_compare:w
4564        \dim_use:N \__dim_eval:w #1 \__prg_compare_error:
4565    }
4566 \cs_new:Npn \__dim_compare:w #1 \__prg_compare_error:
4567    {
4568        \exp_after:wN \if_false: \exp:w \exp_end_continue_f:w
4569            \__dim_compare:wNN #1 ? { = \__dim_compare_end:w \else: } \q_stop
4570    }
4571 \exp_args:Nno \use:nn
4572    { \cs_new:Npn \__dim_compare:wNN #1 }
4573    { \tl_to_str:n {pt} }
4574    #2#3
4575    {
4576        \if_meaning:w = #3
4577            \use:c { __dim_compare_#2:w }
4578        \fi:
4579            #1 pt \exp_stop_f:
4580        \prg_return_false:
4581        \exp_after:wN \use_none_delimit_by_q_stop:w
4582    \fi:
4583    \reverse_if:N \if_dim:w #1 pt #2
4584        \exp_after:wN \__dim_compare:wNN
4585        \dim_use:N \__dim_eval:w #3
4586    }
4587 \cs_new:cpn { __dim_compare_ ! :w }
4588        #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
4589 \cs_new:cpn { __dim_compare_ = :w }
4590        #1 \__dim_eval:w = { #1 \__dim_eval:w }
4591 \cs_new:cpn { __dim_compare_ < :w }
4592        #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
4593 \cs_new:cpn { __dim_compare_ > :w }
4594        #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
4595 \cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \q_stop
4596    { #1 \prg_return_false: \else: \prg_return_true: \fi: }
```

(*End definition for* `\dim_compare:nTF` *and others. These functions are documented on page [78](#).*)

`\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:nn`(TF) as described in l3basics.

```
4597 \cs_new:Npn \dim_case:nnTF #1
```

```
4598      {
4599        \exp:w
4600        \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} }
4601      }
4602 \cs_new:Npn \dim_case:nnT #1#2#3
4603      {
4604        \exp:w
4605        \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { }
4606      }
4607 \cs_new:Npn \dim_case:nnF #1#2
4608      {
4609        \exp:w
4610        \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { }
4611      }
4612 \cs_new:Npn \dim_case:nn #1#2
4613      {
4614        \exp:w
4615        \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { }
4616      }
4617 \cs_new:Npn \__dim_case:nnTF #1#2#3#4
4618      { \__dim_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
4619 \cs_new:Npn \__dim_case:nw #1#2#3
4620      {
4621        \dim_compare:nNnTF {#1} = {#2}
4622          { \__dim_case_end:nw {#3} }
4623          { \__dim_case:nw {#1} }
4624      }
4625 \cs_new_eq:NN \__dim_case_end:nw \__prg_case_end:nw
```

(*End definition for* `\dim_case:nnTF` *and others. These functions are documented on page 79.*)

9.6 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.

```
4626 \cs_set:Npn \dim_while_do:nn #1#2
4627      {
4628        \dim_compare:nT {#1}
4629          {
4630            #2
4631            \dim_while_do:nn {#1} {#2}
4632          }
4633      }
4634 \cs_set:Npn \dim_until_do:nn #1#2
4635      {
4636        \dim_compare:nF {#1}
4637          {
4638            #2
4639            \dim_until_do:nn {#1} {#2}
4640          }
4641      }
4642 \cs_set:Npn \dim_do_while:nn #1#2
4643      {
4644        #2
```

354

```
4645        \dim_compare:nT {#1}
4646          { \dim_do_while:nn {#1} {#2} }
4647      }
4648  \cs_set:Npn \dim_do_until:nn #1#2
4649    {
4650      #2
4651      \dim_compare:nF {#1}
4652        { \dim_do_until:nn {#1} {#2} }
4653    }
```

(*End definition for* `\dim_while_do:nn` *and others. These functions are documented on page 80.*)

`\dim_while_do:nNnn` while_do and do_while functions for dimensions. Same as for the int type only the
`\dim_until_do:nNnn` names have changed.
`\dim_do_while:nNnn`
`\dim_do_until:nNnn`
```
4654  \cs_set:Npn \dim_while_do:nNnn #1#2#3#4
4655    {
4656      \dim_compare:nNnT {#1} #2 {#3}
4657        {
4658          #4
4659          \dim_while_do:nNnn {#1} #2 {#3} {#4}
4660        }
4661    }
4662  \cs_set:Npn \dim_until_do:nNnn #1#2#3#4
4663    {
4664      \dim_compare:nNnF {#1} #2 {#3}
4665        {
4666          #4
4667          \dim_until_do:nNnn {#1} #2 {#3} {#4}
4668        }
4669    }
4670  \cs_set:Npn \dim_do_while:nNnn #1#2#3#4
4671    {
4672      #4
4673      \dim_compare:nNnT {#1} #2 {#3}
4674        { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
4675    }
4676  \cs_set:Npn \dim_do_until:nNnn #1#2#3#4
4677    {
4678      #4
4679      \dim_compare:nNnF {#1} #2 {#3}
4680        { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
4681    }
```

(*End definition for* `\dim_while_do:nNnn` *and others. These functions are documented on page 80.*)

9.7   Using `dim` expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.
```
4682  \cs_new:Npn \dim_eval:n #1
4683    { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }
```

(*End definition for* `\dim_eval:n`. *This function is documented on page 80.*)

`\dim_use:N` Accessing a ⟨*dim*⟩.
`\dim_use:c`
```
4684  \cs_new_eq:NN \dim_use:N \tex_the:D
```

We hand-code this for some speed gain:

```
4685 %\cs_generate_variant:Nn \dim_use:N { c }
4686 \cs_new:Npn \dim_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(*End definition for* `\dim_use:N`*. This function is documented on page 80.*)

`\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`. 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.

```
4687 \cs_new:Npn \dim_to_decimal:n #1
4688   {
4689     \exp_after:wN
4690       \__dim_to_decimal:w \dim_use:N \__dim_eval:w (#1) \__dim_eval_end:
4691   }
4692 \use:x
4693   {
4694     \cs_new:Npn \exp_not:N \__dim_to_decimal:w
4695       ##1 . ##2 \tl_to_str:n { pt }
4696   }
4697     {
4698       \int_compare:nNnTF {#2} > \c_zero
4699         { #1 . #2 }
4700         { #1 }
4701     }
```

(*End definition for* `\dim_to_decimal:n` *and* `\__dim_to_decimal:w`*. These functions are documented on page 81.*)

`\dim_to_decimal_in_bp:n`

Conversion to big points is done using a scaling inside `\__dim_eval:w` as ε-TeX does that using 64-bit precision. Here, 800/803 is the integer fraction for 72/72.27. This is a common case so is hand-coded for accuracy (and speed).

```
4702 \cs_new:Npn \dim_to_decimal_in_bp:n #1
4703   { \dim_to_decimal:n { ( #1 ) * 800 / 803 } }
```

(*End definition for* `\dim_to_decimal_in_bp:n`*. This function is documented on page 81.*)

`\dim_to_decimal_in_sp:n`

Another hard-coded conversion: this one is necessary to avoid things going off-scale.

```
4704 \cs_new:Npn \dim_to_decimal_in_sp:n #1
4705   { \int_eval:n { \__dim_eval:w #1 \__dim_eval_end: } }
```

(*End definition for* `\dim_to_decimal_in_sp:n`*. This function is documented on page 81.*)

`\dim_to_decimal_in_unit:nn`

An analogue of `\dim_ratio:nn` that produces a decimal number as its result, rather than a rational fraction for use within dimension expressions.

```
4706 \cs_new:Npn \dim_to_decimal_in_unit:nn #1#2
4707   {
4708     \dim_to_decimal:n
4709       {
4710         1pt *
4711         \dim_ratio:nn {#1} {#2}
4712       }
4713   }
```

(*End definition for* `\dim_to_decimal_in_unit:nn`*. This function is documented on page 81.*)

`\dim_to_fp:n` Defined in l3fp-convert, documented here.

(*End definition for* `\dim_to_fp:n`. *This function is documented on page 82.*)

9.8 Viewing `dim` variables

`\dim_show:N` Diagnostics.
`\dim_show:c`
```
4714 \cs_new_eq:NN \dim_show:N \__kernel_register_show:N
4715 \cs_generate_variant:Nn \dim_show:N { c }
```

(*End definition for* `\dim_show:N`. *This function is documented on page 82.*)

`\dim_show:n` Diagnostics. We don't use the TeX primitive `\showthe` to show dimension expressions: this gives a more unified output.
```
4716 \cs_new_protected:Npn \dim_show:n
4717   { \__msg_show_wrap:Nn \dim_eval:n }
```

(*End definition for* `\dim_show:n`. *This function is documented on page 82.*)

9.9 Constant dimensions

`\c_zero_dim` Constant dimensions.
`\c_max_dim`
```
4718 \dim_const:Nn \c_zero_dim { 0 pt }
4719 \dim_const:Nn \c_max_dim { 16383.99999 pt }
```

(*End definition for* `\c_zero_dim` *and* `\c_max_dim`. *These variables are documented on page 82.*)

9.10 Scratch dimensions

`\l_tmpa_dim` We provide two local and two global scratch registers, maybe we need more or less.
`\l_tmpb_dim`
`\g_tmpa_dim`
```
4720 \dim_new:N \l_tmpa_dim
```
`\g_tmpb_dim`
```
4721 \dim_new:N \l_tmpb_dim
4722 \dim_new:N \g_tmpa_dim
4723 \dim_new:N \g_tmpb_dim
```

(*End definition for* `\l_tmpa_dim` *and others. These variables are documented on page 82.*)

9.11 Creating and initialising `skip` variables

`\skip_new:N` Allocation of a new internal registers.
`\skip_new:c`
```
4724 ⟨*package⟩
4725 \cs_new_protected:Npn \skip_new:N #1
4726   {
4727     \__chk_if_free_cs:N #1
4728     \cs:w newskip \cs_end: #1
4729   }
4730 ⟨/package⟩
4731 \cs_generate_variant:Nn \skip_new:N { c }
```

(*End definition for* `\skip_new:N`. *This function is documented on page 82.*)

`\skip_const:Nn`
`\skip_const:cn`

Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
4732 \cs_new_protected:Npn \skip_const:Nn #1
4733   {
4734     \skip_new:N #1
4735     \skip_gset:Nn #1
4736   }
4737 \cs_generate_variant:Nn \skip_const:Nn { c }
```

(*End definition for* `\skip_const:Nn`*. This function is documented on page 83.*)

`\skip_zero:N`
`\skip_zero:c`
`\skip_gzero:N`
`\skip_gzero:c`

Reset the register to zero.

```
4738 \cs_new_protected:Npn \skip_zero:N #1 { #1 \c_zero_skip }
4739 \cs_new_protected:Npn \skip_gzero:N { \tex_global:D \skip_zero:N }
4740 \cs_generate_variant:Nn \skip_zero:N  { c }
4741 \cs_generate_variant:Nn \skip_gzero:N { c }
```

(*End definition for* `\skip_zero:N` *and* `\skip_gzero:N`*. These functions are documented on page 83.*)

`\skip_zero_new:N`
`\skip_zero_new:c`
`\skip_gzero_new:N`
`\skip_gzero_new:c`

Create a register if needed, otherwise clear it.

```
4742 \cs_new_protected:Npn \skip_zero_new:N  #1
4743   { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
4744 \cs_new_protected:Npn \skip_gzero_new:N #1
4745   { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
4746 \cs_generate_variant:Nn \skip_zero_new:N  { c }
4747 \cs_generate_variant:Nn \skip_gzero_new:N { c }
```

(*End definition for* `\skip_zero_new:N` *and* `\skip_gzero_new:N`*. These functions are documented on page 83.*)

`\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.

```
4748 \prg_new_eq_conditional:NNn \skip_if_exist:N \cs_if_exist:N
4749   { TF , T , F , p }
4750 \prg_new_eq_conditional:NNn \skip_if_exist:c \cs_if_exist:c
4751   { TF , T , F , p }
```

(*End definition for* `\skip_if_exist:NTF`*. This function is documented on page 83.*)

9.12 Setting skip variables

`\skip_set:Nn`
`\skip_set:cn`
`\skip_gset:Nn`
`\skip_gset:cn`

Much the same as for dimensions.

```
4752 \cs_new_protected:Npn \skip_set:Nn #1#2
4753   { #1 ~ \etex_glueexpr:D #2 \scan_stop: }
4754 \cs_new_protected:Npn \skip_gset:Nn { \tex_global:D \skip_set:Nn }
4755 \cs_generate_variant:Nn \skip_set:Nn  { c }
4756 \cs_generate_variant:Nn \skip_gset:Nn { c }
```

(*End definition for* `\skip_set:Nn` *and* `\skip_gset:Nn`*. These functions are documented on page 83.*)

`\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.

```
4757 \cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
4758 \cs_generate_variant:Nn \skip_set_eq:NN {        c }
4759 \cs_generate_variant:Nn \skip_set_eq:NN { Nc , cc }
4760 \cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
4761 \cs_generate_variant:Nn \skip_gset_eq:NN {        c }
4762 \cs_generate_variant:Nn \skip_gset_eq:NN { Nc , cc }
```

(End definition for `\skip_set_eq:NN` *and* `\skip_gset_eq:NN`*. These functions are documented on page 83.)*

`\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`.

```
4763 \cs_new_protected:Npn \skip_add:Nn #1#2
4764   { \tex_advance:D #1 by \etex_glueexpr:D #2 \scan_stop: }
4765 \cs_new_protected:Npn \skip_gadd:Nn { \tex_global:D \skip_add:Nn }
4766 \cs_generate_variant:Nn \skip_add:Nn  { c }
4767 \cs_generate_variant:Nn \skip_gadd:Nn { c }
4768 \cs_new_protected:Npn \skip_sub:Nn #1#2
4769   { \tex_advance:D #1 by - \etex_glueexpr:D #2 \scan_stop: }
4770 \cs_new_protected:Npn \skip_gsub:Nn { \tex_global:D \skip_sub:Nn }
4771 \cs_generate_variant:Nn \skip_sub:Nn  { c }
4772 \cs_generate_variant:Nn \skip_gsub:Nn { c }
```

(End definition for `\skip_add:Nn` *and others. These functions are documented on page 83.)*

9.13 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.

```
4773 \prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
4774   {
4775     \if_int_compare:w
4776       \__str_if_eq_x:nn { \skip_eval:n { #1 } } { \skip_eval:n { #2 } }
4777       = \c_zero
4778         \prg_return_true:
4779     \else:
4780         \prg_return_false:
4781     \fi:
4782   }
```

(End definition for `\skip_if_eq:nnTF`*. This function is documented on page 84.)*

`\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.

```
4783 \cs_set_protected:Npn \__cs_tmp:w #1
4784   {
4785     \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
4786       {
4787         \exp_after:wN \__skip_if_finite:wwNw
4788         \skip_use:N \etex_glueexpr:D ##1 ; \prg_return_false:
4789         #1 ; \prg_return_true: \q_stop
4790       }
4791     \cs_new:Npn \__skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \q_stop {##3}
4792   }
4793 \exp_args:No \__cs_tmp:w { \tl_to_str:n { fil } }
```

(End definition for `\skip_if_finite:nTF` *and* `\__skip_if_finite:wwNw`*. These functions are documented on page 84.)*

9.14 Using `skip` expressions and variables

`\skip_eval:n` Evaluating a skip expression expandably.

```
4794 \cs_new:Npn \skip_eval:n #1
4795   { \skip_use:N \etex_glueexpr:D #1 \scan_stop: }
```

(*End definition for* `\skip_eval:n`*. This function is documented on page 84.*)

`\skip_use:N` Accessing a ⟨*skip*⟩.
`\skip_use:c`

```
4796 \cs_new_eq:NN \skip_use:N \tex_the:D
4797 %\cs_generate_variant:Nn \skip_use:N { c }
4798 \cs_new:Npn \skip_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(*End definition for* `\skip_use:N`*. This function is documented on page 84.*)

9.15 Inserting skips into the output

`\skip_horizontal:N` Inserting skips.
`\skip_horizontal:c`
`\skip_horizontal:n`
`\skip_vertical:N`
`\skip_vertical:c`
`\skip_vertical:n`

```
4799 \cs_new_eq:NN  \skip_horizontal:N \tex_hskip:D
4800 \cs_new:Npn \skip_horizontal:n #1
4801   { \skip_horizontal:N \etex_glueexpr:D #1 \scan_stop: }
4802 \cs_new_eq:NN  \skip_vertical:N \tex_vskip:D
4803 \cs_new:Npn \skip_vertical:n #1
4804   { \skip_vertical:N \etex_glueexpr:D #1 \scan_stop: }
4805 \cs_generate_variant:Nn \skip_horizontal:N { c }
4806 \cs_generate_variant:Nn \skip_vertical:N { c }
```

(*End definition for* `\skip_horizontal:N` *and others. These functions are documented on page 85.*)

9.16 Viewing `skip` variables

`\skip_show:N` Diagnostics.
`\skip_show:c`

```
4807 \cs_new_eq:NN  \skip_show:N \__kernel_register_show:N
4808 \cs_generate_variant:Nn \skip_show:N { c }
```

(*End definition for* `\skip_show:N`*. This function is documented on page 84.*)

`\skip_show:n` Diagnostics. We don't use the TeX primitive `\showthe` to show skip expressions: this gives a more unified output.

```
4809 \cs_new_protected:Npn \skip_show:n
4810   { \__msg_show_wrap:Nn \skip_eval:n }
```

(*End definition for* `\skip_show:n`*. This function is documented on page 84.*)

9.17 Constant skips

`\c_zero_skip` Skips with no rubber component are just dimensions but need to terminate correctly.
`\c_max_skip`

```
4811 \skip_const:Nn \c_zero_skip { \c_zero_dim }
4812 \skip_const:Nn \c_max_skip { \c_max_dim }
```

(*End definition for* `\c_zero_skip` *and* `\c_max_skip`*. These functions are documented on page 85.*)

9.18 Scratch skips

\l_tmpa_skip We provide two local and two global scratch registers, maybe we need more or less.
\l_tmpb_skip
\g_tmpa_skip 4813 \skip_new:N \l_tmpa_skip
\g_tmpb_skip 4814 \skip_new:N \l_tmpb_skip
4815 \skip_new:N \g_tmpa_skip
4816 \skip_new:N \g_tmpb_skip

(*End definition for* \l_tmpa_skip *and others. These variables are documented on page 85.*)

9.19 Creating and initialising `muskip` variables

\muskip_new:N And then we add muskips.
\muskip_new:c
4817 ⟨*package⟩
4818 *\cs_new_protected:Npn \muskip_new:N #1*
4819 *{*
4820 *__chk_if_free_cs:N #1*
4821 *\cs:w newmuskip \cs_end: #1*
4822 *}*
4823 ⟨/package⟩
4824 \cs_generate_variant:Nn \muskip_new:N { c }

(*End definition for* \muskip_new:N*. This function is documented on page 85.*)

\muskip_const:Nn Contrarily to integer constants, we cannot avoid using a register, even for constants.
\muskip_const:cn
4825 \cs_new_protected:Npn \muskip_const:Nn #1
4826 {
4827 \muskip_new:N #1
4828 \muskip_gset:Nn #1
4829 }
4830 \cs_generate_variant:Nn \muskip_const:Nn { c }

(*End definition for* \muskip_const:Nn*. This function is documented on page 86.*)

\muskip_zero:N Reset the register to zero.
\muskip_zero:c
\muskip_gzero:N 4831 \cs_new_protected:Npn \muskip_zero:N #1
\muskip_gzero:c 4832 { #1 \c_zero_muskip }
4833 \cs_new_protected:Npn \muskip_gzero:N { \tex_global:D \muskip_zero:N }
4834 \cs_generate_variant:Nn \muskip_zero:N { c }
4835 \cs_generate_variant:Nn \muskip_gzero:N { c }

(*End definition for* \muskip_zero:N *and* \muskip_gzero:N*. These functions are documented on page 86.*)

\muskip_zero_new:N Create a register if needed, otherwise clear it.
\muskip_zero_new:c
\muskip_gzero_new:N 4836 \cs_new_protected:Npn \muskip_zero_new:N #1
\muskip_gzero_new:c 4837 { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
4838 \cs_new_protected:Npn \muskip_gzero_new:N #1
4839 { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
4840 \cs_generate_variant:Nn \muskip_zero_new:N { c }
4841 \cs_generate_variant:Nn \muskip_gzero_new:N { c }

(*End definition for* \muskip_zero_new:N *and* \muskip_gzero_new:N*. These functions are documented on page 86.*)

`\muskip_if_exist_p:N`
`\muskip_if_exist_p:c`
`\muskip_if_exist:N`_TF_
`\muskip_if_exist:c`_TF_

Copies of the `cs` functions defined in l3basics.

```
4842 \prg_new_eq_conditional:NNn \muskip_if_exist:N \cs_if_exist:N
4843   { TF , T , F , p }
4844 \prg_new_eq_conditional:NNn \muskip_if_exist:c \cs_if_exist:c
4845   { TF , T , F , p }
```

(*End definition for* `\muskip_if_exist:NTF`. *This function is documented on page* *86.*)

9.20 Setting `muskip` variables

`\muskip_set:Nn`
`\muskip_set:cn`
`\muskip_gset:Nn`
`\muskip_gset:cn`

This should be pretty familiar.

```
4846 \cs_new_protected:Npn \muskip_set:Nn #1#2
4847   { #1 ~ \etex_muexpr:D #2 \scan_stop: }
4848 \cs_new_protected:Npn \muskip_gset:Nn { \tex_global:D \muskip_set:Nn }
4849 \cs_generate_variant:Nn \muskip_set:Nn  { c }
4850 \cs_generate_variant:Nn \muskip_gset:Nn { c }
```

(*End definition for* `\muskip_set:Nn` *and* `\muskip_gset:Nn`. *These functions are documented on page* *86.*)

`\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.

```
4851 \cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
4852 \cs_generate_variant:Nn \muskip_set_eq:NN {        c }
4853 \cs_generate_variant:Nn \muskip_set_eq:NN { Nc , cc }
4854 \cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
4855 \cs_generate_variant:Nn \muskip_gset_eq:NN {        c }
4856 \cs_generate_variant:Nn \muskip_gset_eq:NN { Nc , cc }
```

(*End definition for* `\muskip_set_eq:NN` *and* `\muskip_gset_eq:NN`. *These functions are documented on page* *86.*)

`\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`.

```
4857 \cs_new_protected:Npn \muskip_add:Nn #1#2
4858   { \tex_advance:D #1 by \etex_muexpr:D #2 \scan_stop: }
4859 \cs_new_protected:Npn \muskip_gadd:Nn { \tex_global:D \muskip_add:Nn }
4860 \cs_generate_variant:Nn \muskip_add:Nn  { c }
4861 \cs_generate_variant:Nn \muskip_gadd:Nn { c }
4862 \cs_new_protected:Npn \muskip_sub:Nn #1#2
4863   { \tex_advance:D #1 by - \etex_muexpr:D #2 \scan_stop: }
4864 \cs_new_protected:Npn \muskip_gsub:Nn { \tex_global:D \muskip_sub:Nn }
4865 \cs_generate_variant:Nn \muskip_sub:Nn  { c }
4866 \cs_generate_variant:Nn \muskip_gsub:Nn { c }
```

(*End definition for* `\muskip_add:Nn` *and others. These functions are documented on page* *86.*)

9.21 Using `muskip` expressions and variables

`\muskip_eval:n`

Evaluating a muskip expression expandably.

```
4867 \cs_new:Npn \muskip_eval:n #1
4868   { \muskip_use:N \etex_muexpr:D #1 \scan_stop: }
```

(*End definition for* `\muskip_eval:n`. *This function is documented on page* *87.*)

`\muskip_use:N`
`\muskip_use:c` Accessing a ⟨*muskip*⟩.

```
4869 \cs_new_eq:NN \muskip_use:N \tex_the:D
4870 \cs_generate_variant:Nn \muskip_use:N { c }
```

(*End definition for* `\muskip_use:N`. *This function is documented on page 87.*)

9.22 Viewing `muskip` variables

`\muskip_show:N`
`\muskip_show:c` Diagnostics.

```
4871 \cs_new_eq:NN  \muskip_show:N \__kernel_register_show:N
4872 \cs_generate_variant:Nn \muskip_show:N { c }
```

(*End definition for* `\muskip_show:N`. *This function is documented on page 87.*)

`\muskip_show:n` Diagnostics. We don't use the TEX primitive `\showthe` to show muskip expressions: this gives a more unified output.

```
4873 \cs_new_protected:Npn \muskip_show:n
4874   { \__msg_show_wrap:Nn \muskip_eval:n }
```

(*End definition for* `\muskip_show:n`. *This function is documented on page 87.*)

9.23 Constant muskips

`\c_zero_muskip`
`\c_max_muskip` Constant muskips given by their value.

```
4875 \muskip_const:Nn \c_zero_muskip { 0 mu }
4876 \muskip_const:Nn \c_max_muskip  { 16383.99999 mu }
```

(*End definition for* `\c_zero_muskip` *and* `\c_max_muskip`. *These functions are documented on page 87.*)

9.24 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.

```
4877 \muskip_new:N \l_tmpa_muskip
4878 \muskip_new:N \l_tmpb_muskip
4879 \muskip_new:N \g_tmpa_muskip
4880 \muskip_new:N \g_tmpb_muskip
```

(*End definition for* `\l_tmpa_muskip` *and others. These variables are documented on page 87.*)

```
4881 ⟨/initex | package⟩
```

10   l3tl implementation

```
4882 ⟨*initex | package⟩
```

```
4883 ⟨@@=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.

10.1 Functions

`\tl_new:N`
`\tl_new:c` Creating new token list variables is a case of checking for an existing definition and doing the definition.

```
4884 \cs_new_protected:Npn \tl_new:N #1
4885   {
4886     \__chk_if_free_cs:N #1
4887     \cs_gset_eq:NN #1 \c_empty_tl
4888   }
4889 \cs_generate_variant:Nn \tl_new:N { c }
```

(*End definition for* `\tl_new:N`. *This function is documented on page 89.*)

`\tl_const:Nn` Constants are also easy to generate.
`\tl_const:Nx`
`\tl_const:cn`
`\tl_const:cx`
```
4890 \cs_new_protected:Npn \tl_const:Nn #1#2
4891   {
4892     \__chk_if_free_cs:N #1
4893     \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
4894   }
4895 \cs_new_protected:Npn \tl_const:Nx #1#2
4896   {
4897     \__chk_if_free_cs:N #1
4898     \cs_gset_nopar:Npx #1 {#2}
4899   }
4900 \cs_generate_variant:Nn \tl_const:Nn { c }
4901 \cs_generate_variant:Nn \tl_const:Nx { c }
```

(*End definition for* `\tl_const:Nn`. *This function is documented on page 89.*)

`\tl_clear:N` Clearing a token list variable means setting it to an empty value. Error checking will be
`\tl_clear:c` sorted out by the parent function.
`\tl_gclear:N`
`\tl_gclear:c`
```
4902 \cs_new_protected:Npn \tl_clear:N  #1
4903   { \tl_set_eq:NN #1 \c_empty_tl }
4904 \cs_new_protected:Npn \tl_gclear:N #1
4905   { \tl_gset_eq:NN #1 \c_empty_tl }
4906 \cs_generate_variant:Nn \tl_clear:N  { c }
4907 \cs_generate_variant:Nn \tl_gclear:N { c }
```

(*End definition for* `\tl_clear:N` *and* `\tl_gclear:N`. *These functions are documented on page 89.*)

`\tl_clear_new:N` Clearing a token list variable means setting it to an empty value. Error checking will be
`\tl_clear_new:c` sorted out by the parent function.
`\tl_gclear_new:N`
`\tl_gclear_new:c`
```
4908 \cs_new_protected:Npn \tl_clear_new:N  #1
4909   { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
4910 \cs_new_protected:Npn \tl_gclear_new:N #1
4911   { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
4912 \cs_generate_variant:Nn \tl_clear_new:N  { c }
4913 \cs_generate_variant:Nn \tl_gclear_new:N { c }
```

(*End definition for* `\tl_clear_new:N` *and* `\tl_gclear_new:N`. *These functions are documented on page 90.*)

`\tl_set_eq:NN` For setting token list variables equal to each other.
`\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`
```
4914 \cs_new_eq:NN \tl_set_eq:NN  \cs_set_eq:NN
4915 \cs_new_eq:NN \tl_set_eq:cN  \cs_set_eq:cN
4916 \cs_new_eq:NN \tl_set_eq:Nc  \cs_set_eq:Nc
4917 \cs_new_eq:NN \tl_set_eq:cc  \cs_set_eq:cc
4918 \cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
4919 \cs_new_eq:NN \tl_gset_eq:cN \cs_gset_eq:cN
4920 \cs_new_eq:NN \tl_gset_eq:Nc \cs_gset_eq:Nc
4921 \cs_new_eq:NN \tl_gset_eq:cc \cs_gset_eq:cc
```

(End definition for `\tl_set_eq:NN` *and* `\tl_gset_eq:NN`*. These functions are documented on page 90.)*

`\tl_concat:NNN` Concatenating token lists is easy.
`\tl_concat:ccc`
`\tl_gconcat:NNN`
`\tl_gconcat:ccc`

```
4922 \cs_new_protected:Npn \tl_concat:NNN #1#2#3
4923   { \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
4924 \cs_new_protected:Npn \tl_gconcat:NNN #1#2#3
4925   { \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
4926 \cs_generate_variant:Nn \tl_concat:NNN  { ccc }
4927 \cs_generate_variant:Nn \tl_gconcat:NNN { ccc }
```

(End definition for `\tl_concat:NNN` *and* `\tl_gconcat:NNN`*. These functions are documented on page 90.)*

`\tl_if_exist_p:N` Copies of the `cs` functions defined in l3basics.
`\tl_if_exist_p:c`
`\tl_if_exist:NTF`
`\tl_if_exist:cTF`

```
4928 \prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }
4929 \prg_new_eq_conditional:NNn \tl_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(End definition for `\tl_if_exist:NTF`*. This function is documented on page 90.)*

10.2 Constant token lists

`\c_empty_tl` Never full. We need to define that constant before using `\tl_new:N`.

```
4930 \tl_const:Nn \c_empty_tl { }
```

(End definition for `\c_empty_tl`*. This variable is documented on page 101.)*

`\c_space_tl` A space as a token list (as opposed to as a character).

```
4931 \tl_const:Nn \c_space_tl { ~ }
```

(End definition for `\c_space_tl`*. This variable is documented on page 101.)*

10.3 Adding to token list variables

`\tl_set:Nn`   By using `\exp_not:n` token list variables can contain `#` tokens, which makes the token
`\tl_set:NV`   list registers provided by TeX more or less redundant. The `\tl_set:No` version is done
`\tl_set:Nv`   "by hand" as it is used quite a lot.
`\tl_set:No`
`\tl_set:Nf`
`\tl_set:Nx`
`\tl_set:cn`
`\tl_set:cV`
`\tl_set:cv`
`\tl_set:co`
`\tl_set:cf`
`\tl_set:cx`
`\tl_gset:Nn`
`\tl_gset:NV`
`\tl_gset:Nv`
`\tl_gset:No`
`\tl_gset:Nf`
`\tl_gset:Nx`
`\tl_gset:cn`
`\tl_gset:cV`
`\tl_gset:cv`
`\tl_gset:co`
`\tl_gset:cf`
`\tl_gset:cx`

```
4932 \cs_new_protected:Npn \tl_set:Nn #1#2
4933   { \cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
4934 \cs_new_protected:Npn \tl_set:No #1#2
4935   { \cs_set_nopar:Npx #1 { \exp_not:o {#2} } }
4936 \cs_new_protected:Npn \tl_set:Nx #1#2
4937   { \cs_set_nopar:Npx #1 {#2} }
4938 \cs_new_protected:Npn \tl_gset:Nn #1#2
4939   { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} } }
4940 \cs_new_protected:Npn \tl_gset:No #1#2
4941   { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} } }
4942 \cs_new_protected:Npn \tl_gset:Nx #1#2
4943   { \cs_gset_nopar:Npx #1 {#2} }
4944 \cs_generate_variant:Nn \tl_set:Nn  {         NV , Nv , Nf }
4945 \cs_generate_variant:Nn \tl_set:Nx  { c }
4946 \cs_generate_variant:Nn \tl_set:Nn  { c, co , cV , cv , cf }
4947 \cs_generate_variant:Nn \tl_gset:Nn {         NV , Nv , Nf }
4948 \cs_generate_variant:Nn \tl_gset:Nx { c }
4949 \cs_generate_variant:Nn \tl_gset:Nn { c, co , cV , cv , cf }
```

`\tl_put_left:Nn` Adding to the left is done directly to gain a little performance.
`\tl_put_left:NV`
`\tl_put_left:No`
`\tl_put_left:Nx`
`\tl_put_left:cn`
`\tl_put_left:cV`
`\tl_put_left:co`
`\tl_put_left:cx`
`\tl_gput_left:Nn`
`\tl_gput_left:NV`
`\tl_gput_left:No`
`\tl_gput_left:Nx`
`\tl_gput_left:cn`
`\tl_gput_left:cV`
`\tl_gput_left:co`
`\tl_gput_left:cx`

```
4950 \cs_new_protected:Npn \tl_put_left:Nn #1#2
4951   { \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
4952 \cs_new_protected:Npn \tl_put_left:NV #1#2
4953   { \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
4954 \cs_new_protected:Npn \tl_put_left:No #1#2
4955   { \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
4956 \cs_new_protected:Npn \tl_put_left:Nx #1#2
4957   { \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 } }
4958 \cs_new_protected:Npn \tl_gput_left:Nn #1#2
4959   { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
4960 \cs_new_protected:Npn \tl_gput_left:NV #1#2
4961   { \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
4962 \cs_new_protected:Npn \tl_gput_left:No #1#2
4963   { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
4964 \cs_new_protected:Npn \tl_gput_left:Nx #1#2
4965   { \cs_gset_nopar:Npx #1 { #2 \exp_not:o {#1} } }
4966 \cs_generate_variant:Nn \tl_put_left:Nn  { c }
4967 \cs_generate_variant:Nn \tl_put_left:NV  { c }
4968 \cs_generate_variant:Nn \tl_put_left:No  { c }
4969 \cs_generate_variant:Nn \tl_put_left:Nx  { c }
4970 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
4971 \cs_generate_variant:Nn \tl_gput_left:NV { c }
4972 \cs_generate_variant:Nn \tl_gput_left:No { c }
4973 \cs_generate_variant:Nn \tl_gput_left:Nx { c }
```

`\tl_put_right:Nn` The same on the right.
`\tl_put_right:NV`
`\tl_put_right:No`
`\tl_put_right:Nx`
`\tl_put_right:cn`
`\tl_put_right:cV`
`\tl_put_right:co`
`\tl_put_right:cx`
`\tl_gput_right:Nn`
`\tl_gput_right:NV`
`\tl_gput_right:No`
`\tl_gput_right:Nx`
`\tl_gput_right:cn`
`\tl_gput_right:cV`
`\tl_gput_right:co`
`\tl_gput_right:cx`

```
4974 \cs_new_protected:Npn \tl_put_right:Nn #1#2
4975   { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
4976 \cs_new_protected:Npn \tl_put_right:NV #1#2
4977   { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
4978 \cs_new_protected:Npn \tl_put_right:No #1#2
4979   { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
4980 \cs_new_protected:Npn \tl_put_right:Nx #1#2
4981   { \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 } }
4982 \cs_new_protected:Npn \tl_gput_right:Nn #1#2
4983   { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
4984 \cs_new_protected:Npn \tl_gput_right:NV #1#2
4985   { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
4986 \cs_new_protected:Npn \tl_gput_right:No #1#2
4987   { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
4988 \cs_new_protected:Npn \tl_gput_right:Nx #1#2
4989   { \cs_gset_nopar:Npx #1 { \exp_not:o {#1} #2 } }
4990 \cs_generate_variant:Nn \tl_put_right:Nn  { c }
4991 \cs_generate_variant:Nn \tl_put_right:NV  { c }
4992 \cs_generate_variant:Nn \tl_put_right:No  { c }
4993 \cs_generate_variant:Nn \tl_put_right:Nx  { c }
4994 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
4995 \cs_generate_variant:Nn \tl_gput_right:NV { c }
```

```
4996 \cs_generate_variant:Nn \tl_gput_right:No { c }
4997 \cs_generate_variant:Nn \tl_gput_right:Nx { c }
```

(*End definition for* \tl_put_right:Nn *and* \tl_gput_right:Nn. *These functions are documented on page*
90.)

When used as a package, there is an option to be picky and to check definitions exist.
This part of the process is done now, so that variable types based on tl (for example
clist, seq and prop) will inherit the appropriate definitions. No \tl_map_... yet as
the mechanisms are not fully in place. Thus instead do a more low level set up for a
mapping, as in l3basics.

```
4998 ⟨*package⟩
4999 \tex_ifodd:D \l@expl@check@declarations@bool
5000   \cs_set_protected:Npn \__cs_tmp:w #1
5001     {
5002       \if_meaning:w \q_recursion_tail #1
5003         \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
5004       \fi:
5005       \use:x
5006         {
5007           \cs_set_protected:Npn #1 \exp_not:n { ##1 ##2 }
5008             {
5009               \__chk_if_exist_var:N \exp_not:n {##1}
5010               \exp_not:o { #1 {##1} {##2} }
5011             }
5012         }
5013       \__cs_tmp:w
5014     }
5015   \__cs_tmp:w
5016     \tl_set:Nn  \tl_set:No  \tl_set:Nx
5017     \tl_gset:Nn \tl_gset:No \tl_gset:Nx
5018     \tl_put_left:Nn   \tl_put_left:NV
5019     \tl_put_left:No   \tl_put_left:Nx
5020     \tl_gput_left:Nn  \tl_gput_left:NV
5021     \tl_gput_left:No  \tl_gput_left:Nx
5022     \tl_put_right:Nn  \tl_put_right:NV
5023     \tl_put_right:No  \tl_put_right:Nx
5024     \tl_gput_right:Nn \tl_gput_right:NV
5025     \tl_gput_right:No \tl_gput_right:Nx
5026     \q_recursion_tail \q_recursion_stop
5027 ⟨/package⟩
```

The two set_eq functions are done by hand as the internals there are a bit different.

```
5028 ⟨*package⟩
5029   \cs_set_protected:Npn \tl_set_eq:NN #1#2
5030     {
5031       \__chk_if_exist_var:N #1
5032       \__chk_if_exist_var:N #2
5033       \cs_set_eq:NN #1 #2
5034     }
5035   \cs_set_protected:Npn \tl_gset_eq:NN #1#2
5036     {
5037       \__chk_if_exist_var:N #1
5038       \__chk_if_exist_var:N #2
5039       \cs_gset_eq:NN #1 #2
```

367

```
5040            }
5041 ⟨/package⟩
```

There is also a need to check all three arguments of the `concat` functions: a token list `#2` or `#3` equal to `\scan_stop:` would lead to problems later on.

```
5042 ⟨*package⟩
5043   \cs_set_protected:Npn \tl_concat:NNN #1#2#3
5044     {
5045       \__chk_if_exist_var:N #1
5046       \__chk_if_exist_var:N #2
5047       \__chk_if_exist_var:N #3
5048       \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} }
5049     }
5050   \cs_set_protected:Npn \tl_gconcat:NNN #1#2#3
5051     {
5052       \__chk_if_exist_var:N #1
5053       \__chk_if_exist_var:N #2
5054       \__chk_if_exist_var:N #3
5055       \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} }
5056     }
5057 \tex_fi:D
5058 ⟨/package⟩
```

10.4 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.

```
5059 \tl_const:Nx \c__tl_rescan_marker_tl { : \token_to_str:N : }
```

(*End definition for* `\c__tl_rescan_marker_tl`.)

`\tl_set_rescan:Nnn`
`\tl_set_rescan:Nno`
`\tl_set_rescan:Nnx`
`\tl_set_rescan:cnn`
`\tl_set_rescan:cno`
`\tl_set_rescan:cnx`
`\tl_gset_rescan:Nnn`
`\tl_gset_rescan:Nno`
`\tl_gset_rescan:Nnx`
`\tl_gset_rescan:cnn`
`\tl_gset_rescan:cno`
`\tl_gset_rescan:cnx`
`\tl_rescan:nn`
`\__tl_set_rescan:NNnn`
`\__tl_set_rescan_multi:n`
`\__tl_rescan:w`

These functions use a common auxiliary. After some initial setup explained below, and the user setup `#3` (followed by `\scan_stop:` to be safe), the tokens are rescanned by `\__tl_set_rescan:n` and stored into `\l__tl_internal_a_tl`, then passed to `#1#2` outside the group after expansion. The auxiliary `\__tl_set_rescan:n` is defined later: in the simplest case, this auxiliary calls `\__tl_set_rescan_multi:n`, whose code is included here to help understand the approach.

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 ...
```

The standard solution is to use an x-expanding assignment and set `\everyeof` to `\exp_-not:N` to suppress the error at the end of the file. Since the rescanned tokens should not be expanded, they will be taken as a delimited argument of an auxiliary which wraps them in `\exp_not:n` (in fact `\exp_not:o`, as there is a `\prg_do_nothing:` to avoid losing braces). The delimiter cannot appear within the rescanned token list because it contains twice the same character, with different catcodes.

The difference between single-line and multiple-line files complicates the story, as explained below.

```
5060 \cs_new_protected:Npn \tl_set_rescan:Nnn
5061   { \__tl_set_rescan:NNnn \tl_set:Nn }
```

```
5062 \cs_new_protected:Npn \tl_gset_rescan:Nnn
5063   { \__tl_set_rescan:NNnn \tl_gset:Nn }
5064 \cs_new_protected:Npn \tl_rescan:nn
5065   { \__tl_set_rescan:NNnn \prg_do_nothing: \use:n }
5066 \cs_new_protected:Npn \__tl_set_rescan:NNnn #1#2#3#4
5067   {
5068     \tl_if_empty:nTF {#4}
5069       {
5070         \group_begin:
5071           #3
5072         \group_end:
5073         #1 #2 { }
5074       }
5075       {
5076         \group_begin:
5077           \exp_args:No \etex_everyeof:D { \c__tl_rescan_marker_tl \exp_not:N }
5078           \int_compare:nNnT \tex_endlinechar:D = { 32 }
5079             { \tex_endlinechar:D \c_minus_one }
5080           \tex_newlinechar:D \tex_endlinechar:D
5081           #3 \scan_stop:
5082           \exp_args:No \__tl_set_rescan:n { \tl_to_str:n {#4} }
5083           \exp_args:NNNo
5084         \group_end:
5085         #1 #2 \l__tl_internal_a_tl
5086       }
5087   }
5088 \cs_new_protected:Npn \__tl_set_rescan_multi:n #1
5089   {
5090     \tl_set:Nx \l__tl_internal_a_tl
5091       {
5092         \exp_after:wN \__tl_rescan:w
5093         \exp_after:wN \prg_do_nothing:
5094         \etex_scantokens:D {#1}
5095       }
5096   }
5097 \exp_args:Nno \use:nn
5098   { \cs_new:Npn \__tl_rescan:w #1 } \c__tl_rescan_marker_tl
5099   { \exp_not:o {#1} }
5100 \cs_generate_variant:Nn \tl_set_rescan:Nnn  {     Nno , Nnx }
5101 \cs_generate_variant:Nn \tl_set_rescan:Nnn  { c , cno , cnx }
5102 \cs_generate_variant:Nn \tl_gset_rescan:Nnn {     Nno , Nnx }
5103 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { c , cno }
```

(*End definition for* `\tl_set_rescan:Nnn` *and others. These functions are documented on page 92.*)

__tl_set_rescan:n This function calls `\__tl_set_rescan_multiple:n` or `\__tl_set_rescan_single:nn`
__tl_set_rescan:NnTF `{ ' }` depending on whether its argument is a single-line fragment of code/data or
__tl_set_rescan_single:nn is made of multiple lines by testing for the presence of a `\newlinechar` character. If
__tl_set_rescan_single_aux:nn `\newlinechar` is out of range, the argument is assumed to be a single line.

The case of multiple lines is a straightforward application of `\scantokens` as described above. 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.

For a single line, no `\endlinechar` should be added, so it will be 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), 12 (other) and 13 (active) are accepted, as these are 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). Once a valid character is found, run some code very similar to `\__tl_set_rescan_multi:n`, except that `\__tl_-rescan:w` must be redefined to also remove the additional character (with the appropriate catcode). Getting the delimiter with the right catcode requires using `\scantokens` inside an x-expansion, hence using the previous definition of `\__tl_rescan:w` as well. The odd `\exp_not:N \use:n` ensures that the trailing `\exp_not:N` in `\everyeof` does not prevent the expansion of `\c__tl_rescan_marker_tl`, but rather of a closing brace (this does nothing). If no valid character is found, similar code is ran, and the only difference is that trailing spaces are not preserved (bear in mind that this only happens if no character between 39 and 127 has catcode letter, other or active).

There is also some work to preserve leading spaces: test whether the first character (given by `\str_head:n`, with an extra space to circumvent a limitation of f-expansion) has catcode 10 and add what TeX would add in the middle of a line for any sequence of such characters: a single space with catcode 10 and character code 32.

```
5104  \group_begin:
5105    \tex_catcode:D '\^^@ = 12 \scan_stop:
5106    \cs_new_protected:Npn \__tl_set_rescan:n #1
5107      {
5108        \int_compare:nNnTF \tex_newlinechar:D < \c_zero
5109          { \use_ii:nn }
5110          {
5111            \char_set_lccode:nn { 0 } { \tex_newlinechar:D }
5112            \tex_lowercase:D { \__tl_set_rescan:NnTF ^^@ } {#1}
5113          }
5114          { \__tl_set_rescan_multi:n }
5115          { \__tl_set_rescan_single:nn { ' } }
5116        {#1}
5117      }
5118    \cs_new_protected:Npn \__tl_set_rescan:NnTF #1#2
5119      { \tl_if_in:nnTF {#2} {#1} }
5120    \cs_new_protected:Npn \__tl_set_rescan_single:nn #1
5121      {
5122        \int_compare:nNnTF
5123          { \char_value_catcode:n { '#1 } / \c_three } = \c_four
5124          { \__tl_set_rescan_single_aux:nn {#1} }
5125          {
```

```
5126            \int_compare:nNnTF { '#1 } < { '\~ }
5127              {
5128                \char_set_lccode:nn { 0 } { '#1 + 1 }
5129                \tex_lowercase:D { \__tl_set_rescan_single:nn { ^^@ } }
5130              }
5131              { \__tl_set_rescan_single_aux:nn { } }
5132          }
5133      }
5134  \cs_new_protected:Npn \__tl_set_rescan_single_aux:nn #1#2
5135      {
5136        \tex_endlinechar:D \c_minus_one
5137        \use:x
5138          {
5139            \exp_not:N \use:n
5140              {
5141                \exp_not:n { \cs_set:Npn \__tl_rescan:w ##1 }
5142                \exp_after:wN \__tl_rescan:w
5143                \exp_after:wN \prg_do_nothing:
5144                \etex_scantokens:D {#1}
5145              }
5146            \c__tl_rescan_marker_tl
5147          }
5148          { \exp_not:o {##1} }
5149        \tl_set:Nx \l__tl_internal_a_tl
5150          {
5151            \int_compare:nNnT
5152              {
5153                \char_value_catcode:n
5154                  { \exp_last_unbraced:Nf ' \str_head:n {#2} ~ }
5155              }
5156              = \c_ten { ~ }
5157            \exp_after:wN \__tl_rescan:w
5158            \exp_after:wN \prg_do_nothing:
5159            \etex_scantokens:D { #2 #1 }
5160          }
5161      }
5162  \group_end:
```

(*End definition for* `\__tl_set_rescan:n` *and others.*)

10.5   Modifying token list variables

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 x-type assignment function `\tl_set:Nx` or `\tl_-gset:Nx` 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*⟩.

```
5163  \cs_new_protected:Npn \tl_replace_once:Nnn
5164    { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_wrap:w \tl_set:Nx  }
5165  \cs_new_protected:Npn \tl_greplace_once:Nnn
5166    { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_wrap:w \tl_gset:Nx }
```

```
5167  \cs_new_protected:Npn \tl_replace_all:Nnn
5168    { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_next:w \tl_set:Nx  }
5169  \cs_new_protected:Npn \tl_greplace_all:Nnn
5170    { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_next:w \tl_gset:Nx }
5171  \cs_generate_variant:Nn \tl_replace_once:Nnn  { c }
5172  \cs_generate_variant:Nn \tl_greplace_once:Nnn { c }
5173  \cs_generate_variant:Nn \tl_replace_all:Nnn   { c }
5174  \cs_generate_variant:Nn \tl_greplace_all:Nnn  { c }
```

(*End definition for* `\tl_replace_all:Nnn` *and others. These functions are documented on page 91.*)

`\__tl_replace:NnNNNnn`
`\__tl_replace_auxi:NnnNNNnn`
`\__tl_replace_auxii:nNNNnn`
`\__tl_replace_next:w`
`\__tl_replace_wrap:w`

To implement the actual replacement auxiliary `\__tl_replace_auxii:nNNNnn` we will 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*⟩ will simply be `\q_mark` in the most common situation where neither the ⟨*token list*⟩ nor the ⟨*pattern*⟩ contains `\q_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_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_nil` or `\q_stop` such that it is not equal to `#6`.

The `\__tl_replace_auxi:NnnNNNnn` auxiliary receives {⟨*A*⟩} and {⟨*A*⟩^{n}⟨*B*⟩} as its arguments, initially with $n = 1$. If "⟨*A*⟩ ⟨*A*⟩^{n}⟨*B*⟩ ⟨*A*⟩^{n}⟨*B*⟩" 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.

```
5175  \cs_new_protected:Npn \__tl_replace:NnNNNnn #1#2#3#4#5#6#7
```

```
5176    {
5177      \tl_if_empty:nTF {#6}
5178        {
5179          \__msg_kernel_error:nnx { kernel } { empty-search-pattern }
5180            { \tl_to_str:n {#7} }
5181        }
5182        {
5183          \tl_if_in:onTF { #5 #6 } {#1}
5184            {
5185              \tl_if_in:nnTF {#6} {#1}
5186                { \exp_args:Nc \__tl_replace:NnNNNnn {#2} {#2?} }
5187                {
5188                  \quark_if_nil:nTF {#6}
5189                    { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_stop } }
5190                    { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_nil  } }
5191                }
5192            }
5193            { \__tl_replace_auxii:nNNNnn {#1} }
5194          #3#4#5 {#6} {#7}
5195        }
5196    }
5197 \cs_new_protected:Npn \__tl_replace_auxi:NnnNNNnn #1#2#3
5198    {
5199      \tl_if_in:NnTF #1 { #2 #3 #3 }
5200        { \__tl_replace_auxi:NnnNNNnn #1 { #2 #3 } {#2} }
5201        { \__tl_replace_auxii:nNNNnn { #2 #3 #3 } }
5202    }
```

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 x-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 x-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*⟩).

```
5203 \cs_new_protected:Npn \__tl_replace_auxii:nNNNnn #1#2#3#4#5#6
5204   {
5205     \group_align_safe_begin:
5206     \cs_set:Npn \__tl_replace_wrap:w ##1 #1 ##2
5207       { \exp_not:o { \use_none:nn ##1 } ##2 }
5208     \cs_set:Npx \__tl_replace_next:w ##1 #5
5209       {
5210         \exp_not:N \__tl_replace_wrap:w ##1
5211         \exp_not:n { #1 }
5212         \exp_not:n { \exp_not:n {#6} }
5213         \exp_not:n { #2 { } { } }
5214       }
5215     #3 #4
5216       {
5217         \exp_after:wN \__tl_replace_next:w
5218         \exp_after:wN { \exp_after:wN }
5219         \exp_after:wN { \exp_after:wN }
5220         #4
5221         #1
5222         {
5223           \if_false: { \fi: }
5224           \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
5225         }
5226         #5
5227       }
5228     \group_align_safe_end:
5229   }
5230 \cs_new_eq:NN \__tl_replace_wrap:w ?
5231 \cs_new_eq:NN \__tl_replace_next:w ?
```

(*End definition for* `\__tl_replace:NnNNNnn` *and others.*)

`\tl_remove_once:Nn`
`\tl_remove_once:cn`
`\tl_gremove_once:Nn`
`\tl_gremove_once:cn`

Removal is just a special case of replacement.

```
5232 \cs_new_protected:Npn \tl_remove_once:Nn #1#2
5233   { \tl_replace_once:Nnn #1 {#2} { } }
5234 \cs_new_protected:Npn \tl_gremove_once:Nn #1#2
5235   { \tl_greplace_once:Nnn #1 {#2} { } }
5236 \cs_generate_variant:Nn \tl_remove_once:Nn  { c }
5237 \cs_generate_variant:Nn \tl_gremove_once:Nn { c }
```

(*End definition for* `\tl_remove_once:Nn` *and* `\tl_gremove_once:Nn`*. These functions are documented on page 91.*)

`\tl_remove_all:Nn`
`\tl_remove_all:cn`
`\tl_gremove_all:Nn`
`\tl_gremove_all:cn`

Removal is just a special case of replacement.

```
5238 \cs_new_protected:Npn \tl_remove_all:Nn #1#2
5239   { \tl_replace_all:Nnn #1 {#2} { } }
5240 \cs_new_protected:Npn \tl_gremove_all:Nn #1#2
5241   { \tl_greplace_all:Nnn #1 {#2} { } }
5242 \cs_generate_variant:Nn \tl_remove_all:Nn  { c }
5243 \cs_generate_variant:Nn \tl_gremove_all:Nn { c }
```

(*End definition for* `\tl_remove_all:Nn` *and* `\tl_gremove_all:Nn`*. These functions are documented on page 91.*)

10.6 Token list conditionals

`\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_return:o` is a fast emptyness test, converting its argument to a string (after one expansion) and using the test `\if_meaning:w \q_nil ... \q_nil`.

```
5244 \prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF }
5245   { \__tl_if_empty_return:o { \use_none:n #1 ? } }
5246 \cs_generate_variant:Nn \tl_if_blank_p:n { V }
5247 \cs_generate_variant:Nn \tl_if_blank:nT  { V }
5248 \cs_generate_variant:Nn \tl_if_blank:nF  { V }
5249 \cs_generate_variant:Nn \tl_if_blank:nTF { V }
5250 \cs_generate_variant:Nn \tl_if_blank_p:n { o }
5251 \cs_generate_variant:Nn \tl_if_blank:nT  { o }
5252 \cs_generate_variant:Nn \tl_if_blank:nF  { o }
5253 \cs_generate_variant:Nn \tl_if_blank:nTF { o }
```

(*End definition for* `\tl_if_blank:nTF` *and* `\__tl_if_blank_p:NNw`*. These functions are documented on page 92.*)

`\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).

```
5254 \prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
5255   {
5256     \if_meaning:w #1 \c_empty_tl
5257       \prg_return_true:
5258     \else:
5259       \prg_return_false:
5260     \fi:
5261   }
5262 \cs_generate_variant:Nn \tl_if_empty_p:N { c }
5263 \cs_generate_variant:Nn \tl_if_empty:NT  { c }
5264 \cs_generate_variant:Nn \tl_if_empty:NF  { c }
5265 \cs_generate_variant:Nn \tl_if_empty:NTF { c }
```

(*End definition for* `\tl_if_empty:NTF`*. This function is documented on page 93.*)

`\tl_if_empty_p:n`
`\tl_if_empty_p:V`
`\tl_if_empty:nTF`
`\tl_if_empty:VTF`

Convert the argument to a string: this will be empty if and only if the argument is. Then `\if_meaning:w \q_nil ... \q_nil` is `true` if and only if the string ... is empty. It could be tempting to use `\if_meaning:w \q_nil #1 \q_nil` directly. This fails on a token list starting with `\q_nil` of course but more troubling is the case where argument is a complete conditional such as `\if_true: a \else: b \fi:` because then `\if_true:` is used by `\if_meaning:w`, the test turns out `false`, the `\else:` executes the `false` branch, the `\fi:` ends it and the `\q_nil` at the end starts executing...

```
5266 \prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
5267   {
5268     \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
5269       \tl_to_str:n {#1} \q_nil
5270       \prg_return_true:
5271     \else:
5272       \prg_return_false:
5273     \fi:
5274   }
5275 \cs_generate_variant:Nn \tl_if_empty_p:n { V }
```

```
5276 \cs_generate_variant:Nn \tl_if_empty:nTF { V }
5277 \cs_generate_variant:Nn \tl_if_empty:nT  { V }
5278 \cs_generate_variant:Nn \tl_if_empty:nF  { V }
```

(*End definition for* `\tl_if_empty:nTF`*. This function is documented on page 93.*)

\tl_if_empty_p:o
\tl_if_empty:oTF
__tl_if_empty_return:o

The auxiliary function `\__tl_if_empty_return: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 many places. Note that this works because `\etex_detokenize:D` expands tokens that follow until reading a catcode 1 (begin-group) token.

```
5279 \cs_new:Npn \__tl_if_empty_return:o #1
5280   {
5281     \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
5282       \etex_detokenize:D \exp_after:wN {#1} \q_nil
5283       \prg_return_true:
5284     \else:
5285       \prg_return_false:
5286     \fi:
5287   }
5288 \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F }
5289   { \__tl_if_empty_return:o {#1} }
```

(*End definition for* `\tl_if_empty:oTF` *and* `\__tl_if_empty_return:o`*. These functions are documented on page 93.*)

\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.

```
5290 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF }
5291   {
5292     \if_meaning:w #1 #2
5293       \prg_return_true:
5294     \else:
5295       \prg_return_false:
5296     \fi:
5297   }
5298 \cs_generate_variant:Nn \tl_if_eq_p:NN { Nc , c , cc }
5299 \cs_generate_variant:Nn \tl_if_eq:NNTF { Nc , c , cc }
5300 \cs_generate_variant:Nn \tl_if_eq:NNT  { Nc , c , cc }
5301 \cs_generate_variant:Nn \tl_if_eq:NNF  { Nc , c , cc }
```

(*End definition for* `\tl_if_eq:NNTF`*. This function is documented on page 93.*)

\tl_if_eq:nnTF
\l__tl_internal_a_tl
\l__tl_internal_b_tl

A simple store and compare routine.

```
5302 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F ,  TF }
5303   {
5304     \group_begin:
5305       \tl_set:Nn \l__tl_internal_a_tl {#1}
5306       \tl_set:Nn \l__tl_internal_b_tl {#2}
5307       \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
5308         \group_end:
5309         \prg_return_true:
5310       \else:
5311         \group_end:
```

376

```
5312              \prg_return_false:
5313          \fi:
5314      }
5315 \tl_new:N \l__tl_internal_a_tl
5316 \tl_new:N \l__tl_internal_b_tl
```

(*End definition for* `\tl_if_eq:nnTF`, `\l__tl_internal_a_tl`, *and* `\l__tl_internal_b_tl`*. These functions are documented on page 93.*)

`\tl_if_in:Nn`*TF*
`\tl_if_in:cn`*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`.

```
5317 \cs_new_protected:Npn \tl_if_in:NnT  { \exp_args:No \tl_if_in:nnT  }
5318 \cs_new_protected:Npn \tl_if_in:NnF  { \exp_args:No \tl_if_in:nnF  }
5319 \cs_new_protected:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
5320 \cs_generate_variant:Nn \tl_if_in:NnT { c }
5321 \cs_generate_variant:Nn \tl_if_in:NnF  { c }
5322 \cs_generate_variant:Nn \tl_if_in:NnTF { c }
```

(*End definition for* `\tl_if_in:NnTF`*. This function is documented on page 93.*)

`\tl_if_in:nn`*TF*
`\tl_if_in:Vn`*TF*
`\tl_if_in:on`*TF*
`\tl_if_in:no`*TF*

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:`.

```
5323 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
5324   {
5325     \if_false: { \fi:
5326     \cs_set:Npn \__tl_tmp:w ##1 #2 { }
5327     \tl_if_empty:oTF { \__tl_tmp:w #1 {} {} #2 }
5328       { \prg_return_false: } { \prg_return_true: }
5329     \if_false: } \fi:
5330   }
5331 \cs_generate_variant:Nn \tl_if_in:nnT  { V , o , no }
5332 \cs_generate_variant:Nn \tl_if_in:nnF  { V , o , no }
5333 \cs_generate_variant:Nn \tl_if_in:nnTF { V , o , no }
```

(*End definition for* `\tl_if_in:nnTF`*. This function is documented on page 93.*)

`\tl_if_single_p:N`
`\tl_if_single:N`*TF*

Expand the token list and feed it to `\tl_if_single:n`.

```
5334 \cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
5335 \cs_new:Npn \tl_if_single:NT  { \exp_args:No \tl_if_single:nT  }
5336 \cs_new:Npn \tl_if_single:NF  { \exp_args:No \tl_if_single:nF  }
5337 \cs_new:Npn \tl_if_single:NTF { \exp_args:No \tl_if_single:nTF }
```

(*End definition for* `\tl_if_single:NTF`*. This function is documented on page 93.*)

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, `\tl_to_str:n` 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 will be the trailing `?` and the catcode test yields `false`. If `#1` has a single item, the token will be `^` and the catcode test yields `true`. Otherwise, it will be one of the characters resulting from `\tl_to_str:n`, and the catcode test yields `false`. Note that `\if_catcode:w` takes care of the expansions, and that `\tl_to_str:n` (the `\detokenize` primitive) actually expands tokens until finding a begin-group token.

```
5338 \prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
5339   {
5340     \if_catcode:w ^ \exp_after:wN \__tl_if_single:nnw
5341       \tl_to_str:n \exp_after:wN { \use_none:nn #1 ?? } ^ ? \q_stop
5342     \prg_return_true:
5343     \else:
5344       \prg_return_false:
5345     \fi:
5346   }
5347 \cs_new:Npn \__tl_if_single:nnw #1#2#3 \q_stop {#2}
```

(*End definition for* `\tl_if_single:nTF` *and* `\__tl_if_single:nTF`*. These functions are documented on page 94.*)

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 will always be 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.

```
5348 \cs_new:Npn \tl_case:Nn #1#2
5349   {
5350     \exp:w
5351     \__tl_case:NnTF #1 {#2} { } { }
5352   }
5353 \cs_new:Npn \tl_case:NnT #1#2#3
5354   {
5355     \exp:w
5356     \__tl_case:NnTF #1 {#2} {#3} { }
5357   }
5358 \cs_new:Npn \tl_case:NnF #1#2#3
5359   {
5360     \exp:w
5361     \__tl_case:NnTF #1 {#2} { } {#3}
5362   }
5363 \cs_new:Npn \tl_case:NnTF #1#2
5364   {
5365     \exp:w
5366     \__tl_case:NnTF #1 {#2}
5367   }
5368 \cs_new:Npn \__tl_case:NnTF #1#2#3#4
5369   { \__tl_case:Nw #1 #2 #1 { } \q_mark {#3} \q_mark {#4} \q_stop }
5370 \cs_new:Npn \__tl_case:Nw #1#2#3
```

```
5371   {
5372     \tl_if_eq:NNTF #1 #2
5373       { \__tl_case_end:nw {#3} }
5374       { \__tl_case:Nw #1 }
5375   }
5376 \cs_generate_variant:Nn \tl_case:Nn   { c }
5377 \cs_generate_variant:Nn \tl_case:NnT  { c }
5378 \cs_generate_variant:Nn \tl_case:NnF  { c }
5379 \cs_generate_variant:Nn \tl_case:NnTF { c }
```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then #1 will be the code to insert, #2 will be the *next* case to check on and #3 will be all of the rest of the cases code. That means that #4 will be the `true` branch code, and #5 will be tidy up the spare `\q_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 will be empty, #2 will be the first `\q_mark` and so #4 will be the `false` code (the `true` code is mopped up by #3).

```
5380 \cs_new:Npn \__prg_case_end:nw #1#2#3 \q_mark #4#5 \q_stop
5381   { \exp_end: #1 #4 }
5382 \cs_new_eq:NN \__tl_case_end:nw \__prg_case_end:nw
```

(*End definition for* `\tl_case:NnTF` *and others. These functions are documented on page 94.*)

10.7   Mapping to token lists

\tl_map_function:nN
\tl_map_function:NN
\tl_map_function:cN
__tl_map_function:Nn

Expandable loop macro for token lists. These have the advantage of not needing to test if the argument is empty, because if it is, the stop marker will be read immediately and the loop terminated.

```
5383 \cs_new:Npn \tl_map_function:nN #1#2
5384   {
5385     \__tl_map_function:Nn #2 #1
5386       \q_recursion_tail
5387     \__prg_break_point:Nn \tl_map_break: { }
5388   }
5389 \cs_new:Npn \tl_map_function:NN
5390   { \exp_args:No \tl_map_function:nN }
5391 \cs_new:Npn \__tl_map_function:Nn #1#2
5392   {
5393     \__quark_if_recursion_tail_break:nN {#2} \tl_map_break:
5394     #1 {#2} \__tl_map_function:Nn #1
5395   }
5396 \cs_generate_variant:Nn \tl_map_function:NN { c }
```

(*End definition for* `\tl_map_function:nN`, `\tl_map_function:NN`, *and* `\__tl_map_function:Nn`. *These functions are documented on page 94.*)

\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__prg_map_int` to make them nestable. We can also make use of `\__tl_map_-function:Nn` from before.

```
5397 \cs_new_protected:Npn \tl_map_inline:nn #1#2
5398   {
5399     \int_gincr:N \g__prg_map_int
5400     \cs_gset:cpn { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}
5401     \exp_args:Nc \__tl_map_function:Nn
```

379

```
5402        { __prg_map_ \int_use:N \g__prg_map_int :w }
5403        #1 \q_recursion_tail
5404      \__prg_break_point:Nn \tl_map_break: { \int_gdecr:N \g__prg_map_int }
5405    }
5406  \cs_new_protected:Npn \tl_map_inline:Nn
5407    { \exp_args:No \tl_map_inline:nn }
5408  \cs_generate_variant:Nn \tl_map_inline:Nn { c }
```

(*End definition for* `\tl_map_inline:nn` *and* `\tl_map_inline:Nn`*. These functions are documented on page 94.*)

`\tl_map_variable:nNn`
`\tl_map_variable:NNn`
`\tl_map_variable:cNn`
`\__tl_map_variable:Nnn`

`\tl_map_variable:nNn` ⟨*token list*⟩ ⟨*temp*⟩ ⟨*action*⟩ assigns ⟨*temp*⟩ to each element and executes ⟨*action*⟩.

```
5409  \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
5410    {
5411      \__tl_map_variable:Nnn #2 {#3} #1
5412        \q_recursion_tail
5413      \__prg_break_point:Nn \tl_map_break: { }
5414    }
5415  \cs_new_protected:Npn \tl_map_variable:NNn
5416    { \exp_args:No \tl_map_variable:nNn }
5417  \cs_new_protected:Npn \__tl_map_variable:Nnn #1#2#3
5418    {
5419      \tl_set:Nn #1 {#3}
5420      \__quark_if_recursion_tail_break:NN #1 \tl_map_break:
5421      \use:n {#2}
5422      \__tl_map_variable:Nnn #1 {#2}
5423    }
5424  \cs_generate_variant:Nn \tl_map_variable:NNn { c }
```

(*End definition for* `\tl_map_variable:nNn`*,* `\tl_map_variable:NNn`*, and* `\__tl_map_variable:Nnn`*. These functions are documented on page 95.*)

`\tl_map_break:`
`\tl_map_break:n`

The break statements use the general `\__prg_map_break:Nn`.

```
5425  \cs_new:Npn \tl_map_break:
5426    { \__prg_map_break:Nn \tl_map_break: { } }
5427  \cs_new:Npn \tl_map_break:n
5428    { \__prg_map_break:Nn \tl_map_break: }
```

(*End definition for* `\tl_map_break:` *and* `\tl_map_break:n`*. These functions are documented on page 95.*)

10.8 Using token lists

`\tl_to_str:n` Another name for a primitive: defined in l3basics.

(*End definition for* `\tl_to_str:n`*. This function is documented on page 96.*)

`\tl_to_str:N`
`\tl_to_str:c`

These functions return the replacement text of a token list as a string.

```
5429  \cs_new:Npn \tl_to_str:N #1 { \etex_detokenize:D \exp_after:wN {#1} }
5430  \cs_generate_variant:Nn \tl_to_str:N { c }
```

(*End definition for* `\tl_to_str:N`*. This function is documented on page 96.*)

\tl_use:N Token lists which are simply not defined will give a clear TeX error here. No such luck
\tl_use:c for ones equal to \scan_stop: so instead a test is made and if there is an issue an error
is forced.

```
5431 \cs_new:Npn \tl_use:N #1
5432   {
5433     \tl_if_exist:NTF #1 {#1}
5434       {
5435         \__msg_kernel_expandable_error:nnn
5436           { kernel } { bad-variable } {#1}
5437       }
5438   }
5439 \cs_generate_variant:Nn \tl_use:N { c }
```

(*End definition for* \tl_use:N. *This function is documented on page* *96.*)

10.9   Working with the contents of token lists

\tl_count:n Count number of elements within a token list or token list variable. Brace groups within
\tl_count:V the list are read as a single element. Spaces are ignored. __tl_count:n grabs the
\tl_count:o element and replaces it by +1. The 0 ensures that it works on an empty list.
\tl_count:N
\tl_count:c
__tl_count:n

```
5440 \cs_new:Npn \tl_count:n #1
5441   {
5442     \int_eval:n
5443       { 0 \tl_map_function:nN {#1} \__tl_count:n }
5444   }
5445 \cs_new:Npn \tl_count:N #1
5446   {
5447     \int_eval:n
5448       { 0 \tl_map_function:NN #1 \__tl_count:n }
5449   }
5450 \cs_new:Npn \__tl_count:n #1 { + \c_one }
5451 \cs_generate_variant:Nn \tl_count:n { V , o }
5452 \cs_generate_variant:Nn \tl_count:N { c }
```

(*End definition for* \tl_count:n, \tl_count:N, *and* __tl_count:n. *These functions are documented*
on page *96.*)

\tl_reverse_items:n Reversal of a token list is done by taking one item at a time and putting it after \q_stop.
__tl_reverse_items:nwNwn
__tl_reverse_items:wn

```
5453 \cs_new:Npn \tl_reverse_items:n #1
5454   {
5455     \__tl_reverse_items:nwNwn #1 ?
5456       \q_mark \__tl_reverse_items:nwNwn
5457       \q_mark \__tl_reverse_items:wn
5458       \q_stop { }
5459   }
5460 \cs_new:Npn \__tl_reverse_items:nwNwn #1 #2 \q_mark #3 #4 \q_stop #5
5461   {
5462     #3 #2
5463       \q_mark \__tl_reverse_items:nwNwn
5464       \q_mark \__tl_reverse_items:wn
5465       \q_stop { {#1} #5 }
5466   }
5467 \cs_new:Npn \__tl_reverse_items:wn #1 \q_stop #2
5468   { \exp_not:o { \use_none:nn #2 } }
```

381

(End definition for `\tl_reverse_items:n` *,* `\__tl_reverse_items:nwNwn` *, and* `\__tl_reverse_items:wn` *. These functions are documented on page [97](#).)*

`\tl_trim_spaces:n`
`\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 `\q_mark`, and whose second argument is a ⟨*continuation*⟩, which will receive as a braced argument `\use_none:n \q_-mark` ⟨*trimmed token list*⟩. In the case at hand, we take `\exp_not:o` as our continuation, so that space trimming will behave correctly within an x-type expansion.

```
5469 \cs_new:Npn \tl_trim_spaces:n #1
5470   { \__tl_trim_spaces:nn { \q_mark #1 } \exp_not:o }
5471 \cs_new_protected:Npn \tl_trim_spaces:N #1
5472   { \tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
5473 \cs_new_protected:Npn \tl_gtrim_spaces:N #1
5474   { \tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
5475 \cs_generate_variant:Nn \tl_trim_spaces:N  { c }
5476 \cs_generate_variant:Nn \tl_gtrim_spaces:N { c }
```

(End definition for `\tl_trim_spaces:n` *,* `\tl_trim_spaces:N` *, and* `\tl_gtrim_spaces:N` *. These functions are documented on page [97](#).)*

`\__tl_trim_spaces:nn`
`\__tl_trim_spaces_auxi:w`
`\__tl_trim_spaces_auxii:w`
`\__tl_trim_spaces_auxiii:w`
`\__tl_trim_spaces_auxiv:w`

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 `\q_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 ␣ `\q_nil` matches the one present in the definition of `\tl_trim_spacs:n`. Then `\__tl_trim_spaces_auxiv:w` puts the token list into a group, with `\use_none:n` placed there to gobble a lingering `\q_mark`, and feeds this to the ⟨*continuation*⟩.

```
5477 \cs_set:Npn \__tl_tmp:w #1
5478   {
5479     \cs_new:Npn \__tl_trim_spaces:nn ##1
5480       {
5481         \__tl_trim_spaces_auxi:w
5482           ##1
5483           \q_nil
5484           \q_mark #1 { }
5485           \q_mark \__tl_trim_spaces_auxii:w
5486           \__tl_trim_spaces_auxiii:w
5487           #1 \q_nil
5488           \__tl_trim_spaces_auxiv:w
5489         \q_stop
5490       }
5491     \cs_new:Npn \__tl_trim_spaces_auxi:w ##1 \q_mark #1 ##2 \q_mark ##3
5492       {
5493         ##3
5494         \__tl_trim_spaces_auxi:w
5495         \q_mark
5496         ##2
5497         \q_mark #1 {##1}
5498       }
5499     \cs_new:Npn \__tl_trim_spaces_auxii:w
```

382

```
5500            \__tl_trim_spaces_auxi:w \q_mark \q_mark ##1
5501          {
5502            \__tl_trim_spaces_auxiii:w
5503            ##1
5504          }
5505        \cs_new:Npn \__tl_trim_spaces_auxiii:w ##1 #1 \q_nil ##2
5506          {
5507            ##2
5508            ##1 \q_nil
5509            \__tl_trim_spaces_auxiii:w
5510          }
5511        \cs_new:Npn \__tl_trim_spaces_auxiv:w ##1 \q_nil ##2 \q_stop ##3
5512          { ##3 { \use_none:n ##1 } }
5513      }
5514    \__tl_tmp:w { ~ }
```

(*End definition for* `\__tl_trim_spaces:nn` *and others.*)

10.10   Token by token changes

`\q__tl_act_mark`
`\q__tl_act_stop`

The `\tl_act` functions may be applied to any token list. Hence, we use two private quarks, to allow any token, even quarks, in the token list.Only `\q__tl_act_mark` and `\q__tl_act_stop` may not appear in the token lists manipulated by `\__tl_act:NNNnn` functions. The quarks are effectively defined in l3quark.

(*End definition for* `\q__tl_act_mark` *and* `\q__tl_act_stop`.*)

`\__tl_act:NNNnn`
`\__tl_act_output:n`
`\__tl_act_reverse_output:n`
`\__tl_act_loop:w`
`\__tl_act_normal:NwnNNN`
`\__tl_act_group:nwnNNN`
`\__tl_act_space:wwnNNN`
`\__tl_act_end:w`

To help control the expansion, `\__tl_act:NNNnn` should always be proceeded by `\exp:w` and ends by producing `\exp_end:` once the result has been obtained. Then loop over tokens, groups, and spaces in #5. The marker `\q__tl_act_mark` is used both to avoid losing outer braces and to detect the end of the token list more easily. The result is stored as an argument for the dummy function `\__tl_act_result:n`.

```
5515    \cs_new:Npn \__tl_act:NNNnn #1#2#3#4#5
5516      {
5517        \group_align_safe_begin:
5518        \__tl_act_loop:w #5 \q__tl_act_mark \q__tl_act_stop
5519        {#4} #1 #2 #3
5520        \__tl_act_result:n { }
5521      }
```

In the loop, we check how the token list begins and act accordingly. In the "normal" case, we may have reached `\q__tl_act_mark`, 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 a group or with a space. Some extra work is needed to make `\__tl_act_space:wwnNNN` gobble the space.

```
5522    \cs_new:Npn \__tl_act_loop:w #1 \q__tl_act_stop
5523      {
5524        \tl_if_head_is_N_type:nTF {#1}
5525          { \__tl_act_normal:NwnNNN }
5526          {
5527            \tl_if_head_is_group:nTF {#1}
5528              { \__tl_act_group:nwnNNN }
5529              { \__tl_act_space:wwnNNN }
```

```
5530            }
5531        #1 \q__tl_act_stop
5532      }
5533 \cs_new:Npn \__tl_act_normal:NwnNNN #1 #2 \q__tl_act_stop #3#4
5534    {
5535      \if_meaning:w \q__tl_act_mark #1
5536        \exp_after:wN \__tl_act_end:wn
5537      \fi:
5538      #4 {#3} #1
5539      \__tl_act_loop:w #2 \q__tl_act_stop
5540      {#3} #4
5541    }
5542 \cs_new:Npn \__tl_act_end:wn #1 \__tl_act_result:n #2
5543    { \group_align_safe_end: \exp_end: #2 }
5544 \cs_new:Npn \__tl_act_group:nwnNNN #1 #2 \q__tl_act_stop #3#4#5
5545    {
5546      #5 {#3} {#1}
5547      \__tl_act_loop:w #2 \q__tl_act_stop
5548      {#3} #4 #5
5549    }
5550 \exp_last_unbraced:NNo
5551    \cs_new:Npn \__tl_act_space:wwnNNN \c_space_tl #1 \q__tl_act_stop #2#3#4#5
5552    {
5553      #5 {#2}
5554      \__tl_act_loop:w #1 \q__tl_act_stop
5555      {#2} #3 #4 #5
5556    }
```

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.

```
5557 \cs_new:Npn \__tl_act_output:n #1 #2 \__tl_act_result:n #3
5558    { #2 \__tl_act_result:n { #3 #1 } }
5559 \cs_new:Npn \__tl_act_reverse_output:n #1 #2 \__tl_act_result:n #3
5560    { #2 \__tl_act_result:n { #1 #3 } }
```

(*End definition for* `\__tl_act:NNNnn` *and others.*)

\tl_reverse:n
\tl_reverse:o
\tl_reverse:V
__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:NNNnn`. 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. All of the internal functions here drop one argument: this is needed by `\__tl_act:NNNnn` when changing case (to record which direction the change is in), but not when reversing the tokens.

```
5561 \cs_new:Npn \tl_reverse:n #1
5562    {
5563      \etex_unexpanded:D \exp_after:wN
5564        {
5565          \exp:w
5566          \__tl_act:NNNnn
5567            \__tl_reverse_normal:nN
5568            \__tl_reverse_group_preserve:nn
5569            \__tl_reverse_space:n
5570            { }
5571            {#1}
```

384

```
5572          }
5573        }
5574  \cs_generate_variant:Nn \tl_reverse:n { o , V }
5575  \cs_new:Npn \__tl_reverse_normal:nN #1#2
5576    { \__tl_act_reverse_output:n {#2} }
5577  \cs_new:Npn \__tl_reverse_group_preserve:nn #1#2
5578    { \__tl_act_reverse_output:n { {#2} } }
5579  \cs_new:Npn \__tl_reverse_space:n #1
5580    { \__tl_act_reverse_output:n { ~ } }
```

(*End definition for* `\tl_reverse:n` *and others. These functions are documented on page* 97.)

<div style="float:left">

`\tl_reverse:N`
`\tl_reverse:c`
`\tl_greverse:N`
`\tl_greverse:c`

</div>

This reverses the list, leaving `\exp_stop_f:` in front, which stops the f-expansion.

```
5581  \cs_new_protected:Npn \tl_reverse:N #1
5582    { \tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
5583  \cs_new_protected:Npn \tl_greverse:N #1
5584    { \tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
5585  \cs_generate_variant:Nn \tl_reverse:N  { c }
5586  \cs_generate_variant:Nn \tl_greverse:N { c }
```

(*End definition for* `\tl_reverse:N` *and* `\tl_greverse:N`. *These functions are documented on page* 97.)

10.11   The first token from a token list

<div style="float:left">

`\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_tail:N`
`\tl_tail:n`
`\tl_tail:V`
`\tl_tail:v`
`\tl_tail:f`

</div>

Finding the head of a token list expandably will always strip braces, which is fine as this is consistent with for example mapping to 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. Using a marker, we can see if what we are grabbing is exactly the marker, or there is anything else to deal with. Is there is, there is a loop. If not, tidy up and leave the item in the output stream. More detail in http://tex.stackexchange.com/a/70168.

```
5587  \cs_new:Npn \tl_head:n #1
5588    {
5589      \etex_unexpanded:D
5590        \if_false: { \fi: \__tl_head_auxi:nw #1 { } \q_stop }
5591    }
5592  \cs_new:Npn \__tl_head_auxi:nw #1#2 \q_stop
5593    {
5594      \exp_after:wN \__tl_head_auxii:n \exp_after:wN {
5595        \if_false: } \fi: {#1}
5596    }
5597  \cs_new:Npn \__tl_head_auxii:n #1
5598    {
5599      \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
5600        \tl_to_str:n \exp_after:wN { \use_none:n #1 } \q_nil
5601        \exp_after:wN \use_i:nn
5602      \else:
5603        \exp_after:wN \use_ii:nn
5604      \fi:
5605        {#1}
5606        { \if_false: { \fi: \__tl_head_auxi:nw #1 } }
```

```
5607   }
5608 \cs_generate_variant:Nn \tl_head:n { V , v , f }
5609 \cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
5610 \cs_new:Npn \tl_head:N { \exp_args:No \tl_head:n }
```

To corrected 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
```

will give the wrong result for \tl_tail:n { a { bc } } (the braces will 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.

```
5611 \cs_new:Npn \tl_tail:n #1
5612   {
5613     \etex_unexpanded:D
5614       \tl_if_blank:nTF {#1}
5615         { { } }
5616         { \exp_after:wN { \use_none:n #1 } }
5617   }
5618 \cs_generate_variant:Nn \tl_tail:n { V , v , f }
5619 \cs_new:Npn \tl_tail:N { \exp_args:No \tl_tail:n }
```

(*End definition for* \tl_head:N *and others. These functions are documented on page 98.*)

\tl_if_head_eq_meaning_p:nN
\tl_if_head_eq_meaning:nN*TF*
\tl_if_head_eq_charcode_p:nN
\tl_if_head_eq_charcode:nN*TF*
\tl_if_head_eq_charcode_p:fN
\tl_if_head_eq_charcode:fN*TF*
\tl_if_head_eq_catcode_p:nN
\tl_if_head_eq_catcode:nN*TF*

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 will result in \tl_head:w leaving two tokens: ? which is taken in the \if_charcode:w test, and \use_none:nn, which ensures that \prg_return_false: is returned regardless of whether the charcode test was **true** or **false**.

```
5620 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
5621   {
5622     \if_charcode:w
5623         \exp_not:N #2
5624         \tl_if_head_is_N_type:nTF { #1 ? }
5625           {
5626             \exp_after:wN \exp_not:N
```

```
5627                \tl_head:w #1 { ? \use_none:nn } \q_stop
5628              }
5629            { \str_head:n {#1} }
5630        \prg_return_true:
5631      \else:
5632        \prg_return_false:
5633      \fi:
5634    }
5635 \cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN { f }
5636 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF { f }
5637 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT  { f }
5638 \cs_generate_variant:Nn \tl_if_head_eq_charcode:nNF  { 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 `\prg_return_-true:` and `\else:` with `\use_none:nn` in case the catcode test with the (arbitrarily chosen) `?` is `true`.

```
5639 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
5640    {
5641      \if_catcode:w
5642          \exp_not:N #2
5643          \tl_if_head_is_N_type:nTF { #1 ? }
5644            {
5645              \exp_after:wN \exp_not:N
5646              \tl_head:w #1 { ? \use_none:nn } \q_stop
5647            }
5648            {
5649              \tl_if_head_is_group:nTF {#1}
5650                { \c_group_begin_token }
5651                { \c_space_token }
5652            }
5653        \prg_return_true:
5654      \else:
5655        \prg_return_false:
5656      \fi:
5657    }
```

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 the usual `\prg_return_true:` and `\else:`. 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.

```
5658 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
5659    {
5660      \tl_if_head_is_N_type:nTF { #1 ? }
5661        { \__tl_if_head_eq_meaning_normal:nN }
5662        { \__tl_if_head_eq_meaning_special:nN }
5663      {#1} #2
5664    }
5665 \cs_new:Npn \__tl_if_head_eq_meaning_normal:nN #1 #2
```

```
5666     {
5667       \exp_after:wN \if_meaning:w
5668         \tl_head:w #1 { ?? \use_none:nnn } \q_stop #2
5669         \prg_return_true:
5670       \else:
5671         \prg_return_false:
5672       \fi:
5673     }
5674   \cs_new:Npn \__tl_if_head_eq_meaning_special:nN #1 #2
5675     {
5676       \if_charcode:w \str_head:n {#1} \exp_not:N #2
5677         \exp_after:wN \use:n
5678       \else:
5679         \prg_return_false:
5680         \exp_after:wN \use_none:n
5681       \fi:
5682       {
5683         \if_catcode:w \exp_not:N #2
5684                       \tl_if_head_is_group:nTF {#1}
5685                         { \c_group_begin_token }
5686                         { \c_space_token }
5687           \prg_return_true:
5688         \else:
5689           \prg_return_false:
5690         \fi:
5691       }
5692     }
```

(*End definition for* \tl_if_head_eq_meaning:nTF *and others. These functions are documented on page* *99.*)

\tl_if_head_is_N_type_p:n
\tl_if_head_is_N_type:nTF
__tl_if_head_is_N_type:w

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, the line involving __tl_if_head_is_N_type:w produces ^ (and otherwise nothing). In the third case (begin-group token), the lines involving \exp_after:wN produce a single closing brace. The category code test is thus true exactly in the fourth case, which is what we want. One cannot optimize by moving one of the * to the beginning: if #1 contains primitive conditionals, all of its occurrences must be dealt with before the \if_catcode:w tries to skip the true branch of the conditional.

```
5693   \prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
5694     {
5695       \if_catcode:w
5696         \if_false: { \fi: \__tl_if_head_is_N_type:w ? #1 ~ }
5697         \exp_after:wN \use_none:n
5698           \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
5699         * *
5700         \prg_return_true:
5701       \else:
5702         \prg_return_false:
5703       \fi:
5704     }
5705   \cs_new:Npn \__tl_if_head_is_N_type:w #1 ~
5706     {
5707       \tl_if_empty:oTF { \use_none:n #1 } { ^ } { }
```

388

```
5708        \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
5709    }
```

(*End definition for* `\tl_if_head_is_N_type:nTF` *and* `\__tl_if_head_is_N_type:w`*. These functions are documented on page 100.*)

`\tl_if_head_is_group_p:n`
`\tl_if_head_is_group:nTF`

Pass the first token of #1 through `\token_to_str:N`, then check for the brace balance. The extra `?` caters for an empty argument.[8]

```
5710 \prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
5711    {
5712        \if_catcode:w
5713            \exp_after:wN \use_none:n
5714                \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
5715            * *
5716        \prg_return_false:
5717    \else:
5718        \prg_return_true:
5719    \fi:
5720    }
```

(*End definition for* `\tl_if_head_is_group:nTF`*. This function is documented on page 99.*)

`\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 `?#1?~`. If that is a single `?` 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 `\exp:w` and `\exp_end:` ensure that the result of a single step of expansion directly yields a balanced token list (no trailing closing brace).

```
5721 \prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
5722    {
5723        \exp:w \if_false: { \fi:
5724            \__tl_if_head_is_space:w ? #1 ? ~ }
5725    }
5726 \cs_new:Npn \__tl_if_head_is_space:w #1 ~
5727    {
5728        \tl_if_empty:oTF { \use_none:n #1 }
5729            { \exp_after:wN \exp_end: \exp_after:wN \prg_return_true: }
5730            { \exp_after:wN \exp_end: \exp_after:wN \prg_return_false: }
5731        \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
5732    }
```

(*End definition for* `\tl_if_head_is_space:nTF` *and* `\__tl_if_head_is_space:w`*. These functions are documented on page 100.*)

10.12 Using a single item

`\tl_item:nn`
`\tl_item:Nn`
`\tl_item:cn`
`\__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 `\quark_if_recursion_tail_-stop:n` terminates the loop, and returns nothing at all.

---

[8]Bruno: this could be made faster, but we don't: if we hope to ever have an e-type argument, we need all brace "tricks" to happen in one step of expansion, keeping the token list brace balanced at all times.

```
5733 \cs_new:Npn \tl_item:nn #1#2
5734   {
5735     \exp_args:Nf \__tl_item:nn
5736       {
5737         \int_eval:n
5738           {
5739             \int_compare:nNnT {#2} < \c_zero
5740               { \tl_count:n {#1} + \c_one + }
5741             #2
5742           }
5743       }
5744     #1
5745     \q_recursion_tail
5746     \__prg_break_point:
5747   }
5748 \cs_new:Npn \__tl_item:nn #1#2
5749   {
5750     \__quark_if_recursion_tail_break:nN {#2} \__prg_break:
5751     \int_compare:nNnTF {#1} = \c_one
5752       { \__prg_break:n { \exp_not:n {#2} } }
5753       { \exp_args:Nf \__tl_item:nn { \int_eval:n { #1 - 1 } } }
5754   }
5755 \cs_new:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
5756 \cs_generate_variant:Nn \tl_item:Nn { c }
```

(*End definition for* `\tl_item:nn` *,* `\tl_item:Nn` *, and* `\__tl_item:nn` *. These functions are documented on page 100.*)

10.13   Viewing token lists

`\tl_show:N` Showing token list variables is done after checking that the variable is defined (see `\__-`
`\tl_show:c`   `kernel_register_show:N`).

```
5757 \cs_new_protected:Npn \tl_show:N #1
5758   {
5759     \__msg_show_variable:NNNnn #1 \tl_if_exist:NTF ? { }
5760       { > ~ \token_to_str:N #1 = \tl_to_str:N #1 }
5761   }
5762 \cs_generate_variant:Nn \tl_show:N { c }
```

(*End definition for* `\tl_show:N` *. This function is documented on page 100.*)

`\tl_show:n` The `\__msg_show_wrap:n` internal function performs line-wrapping and shows the result
using the `\etex_showtokens:D` primitive. Since `\tl_to_str:n` is expanded within the
line-wrapping code, the escape character is always a backslash.

```
5763 \cs_new_protected:Npn \tl_show:n #1
5764   { \__msg_show_wrap:n { > ~ \tl_to_str:n {#1} } }
```

(*End definition for* `\tl_show:n` *. This function is documented on page 100.*)

10.14   Scratch token lists

`\g_tmpa_tl` Global temporary token list variables. They are supposed to be set and used immediately,
`\g_tmpb_tl` with no delay between the definition and the use because you can't count on other macros
not to redefine them from under you.

```
5765 \tl_new:N \g_tmpa_tl
5766 \tl_new:N \g_tmpb_tl
```

(*End definition for* \g_tmpa_tl *and* \g_tmpb_tl*. These variables are documented on page 101.*)

\l_tmpa_tl   These are local temporary token list variables. Be sure not to assume that the value you
\l_tmpb_tl   put into them will survive for long—see discussion above.

```
5767 \tl_new:N \l_tmpa_tl
5768 \tl_new:N \l_tmpb_tl
```

(*End definition for* \l_tmpa_tl *and* \l_tmpb_tl*. These variables are documented on page 101.*)

10.15   Deprecated functions

\tl_to_lowercase:n   For removal after 2017-12-31.
\tl_to_uppercase:n
```
5769 \cs_new_protected:Npn \tl_to_lowercase:n #1
5770   { \tex_lowercase:D {#1} }
5771 \cs_new_protected:Npn \tl_to_uppercase:n #1
5772   { \tex_uppercase:D {#1} }
```

(*End definition for* \tl_to_lowercase:n *and* \tl_to_uppercase:n*.*)

```
5773 ⟨/initex | package⟩
```

11   **l3str** implementation

```
5774 ⟨*initex | package⟩
```

```
5775 ⟨@@=str⟩
```

11.1   Creating and setting string variables

\str_new:N   A string is simply a token list. The full mapping system isn't set up yet so do things by
\str_new:c   hand.
\str_use:N
\str_use:c
```
5776 \group_begin:
5777   \cs_set_protected:Npn \__str_tmp:n #1
5778     {
5779       \tl_if_blank:nF {#1}
5780         {
5781           \cs_new_eq:cc { str_ #1 :N } { tl_ #1 :N }
5782           \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :N } { c }
5783           \__str_tmp:n
5784         }
5785     }
5786   \__str_tmp:n
5787     { new }
5788     { use }
5789     { clear }
5790     { gclear }
5791     { clear_new }
5792     { gclear_new }
5793     { }
5794 \group_end:
5795 \cs_new_eq:NN \str_set_eq:NN \tl_set_eq:NN
5796 \cs_new_eq:NN \str_gset_eq:NN \tl_gset_eq:NN
```
\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

```
5797  \cs_generate_variant:Nn \str_set_eq:NN  { c , Nc , cc }
5798  \cs_generate_variant:Nn \str_gset_eq:NN { c , Nc , cc }
```

(*End definition for* `\str_new:N` *and others. These functions are documented on page 102.*)

<div style="display:flex">
<div>

\str_set:Nn
\str_set:Nx
\str_set:cn
\str_set:cx
\str_gset:Nn
\str_gset:Nx
\str_gset:cn
\str_gset:cx
\str_const:Nn
\str_const:Nx
\str_const:cn
\str_const:cx
\str_put_left:Nn
\str_put_left:Nx
\str_put_left:cn
\str_put_left:cx
\str_gput_left:Nn
\str_gput_left:Nx
\str_gput_left:cn
\str_gput_left:cx
\str_put_right:Nn
\str_put_right:Nx
\str_put_right:cn
\str_put_right:cx
\str_gput_right:Nn
\str_gput_right:Nx
\str_gput_right:cn
\str_gput_right:cx
\str_if_empty_p:N
\str_if_empty_p:c
\str_if_empty:NTF
\str_if_empty:cTF
\str_if_exist_p:N
\str_if_exist_p:c
\str_if_exist:NTF
\str_if_exist:cTF
__str_if_eq_x:nn
__str_escape_x:n

</div>
<div>

Simply convert the token list inputs to ⟨*strings*⟩.

```
5799  \group_begin:
5800    \cs_set_protected:Npn \__str_tmp:n #1
5801      {
5802        \tl_if_blank:nF {#1}
5803          {
5804            \cs_new_protected:cpx { str_ #1 :Nn } ##1##2
5805              { \exp_not:c { tl_ #1 :Nx } ##1 { \exp_not:N \tl_to_str:n {##2} } }
5806            \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :Nn } { Nx , cn , cx }
5807            \__str_tmp:n
5808          }
5809      }
5810    \__str_tmp:n
5811      { set }
5812      { gset }
5813      { const }
5814      { put_left }
5815      { gput_left }
5816      { put_right }
5817      { gput_right }
5818      { }
5819  \group_end:
```

(*End definition for* `\str_set:Nn` *and others. These functions are documented on page 103.*)

11.2 String comparisons

More copy-paste!

```
5820  \prg_new_eq_conditional:NNn \str_if_exist:N \tl_if_exist:N { p , T , F , TF }
5821  \prg_new_eq_conditional:NNn \str_if_exist:c \tl_if_exist:c { p , T , F , TF }
5822  \prg_new_eq_conditional:NNn \str_if_empty:N \tl_if_empty:N { p , T , F , TF }
5823  \prg_new_eq_conditional:NNn \str_if_empty:c \tl_if_empty:c { p , T , F , TF }
```

(*End definition for* `\str_if_empty:NTF` *and* `\str_if_exist:NTF`. *These functions are documented on page 104.*)

String comparisons rely on the primitive `\(pdf)strcmp` if available: LuaTeX does not have it, so emulation is required. As the net result is that we do not *always* use the primitive, the correct approach is to wrap up in a function with defined behaviour. That's done by providing a wrapper and then redefining in the LuaTeX case. Note that the necessary Lua code is covered in l3boostrap: long-term this may need to go into a separate Lua file, but at present it's somewhere that spaces are not skipped for ease-of-input. The need to detokenize and force expansion of input arises from the case where a `#` token is used in the input, *e.g.* `\__str_if_eq_x:nn {#} { \tl_to_str:n {#} }`, which otherwise will fail as `\luatex_luaescapestring:D` does not double such tokens.

```
5824  \cs_new:Npn \__str_if_eq_x:nn #1#2 { \pdftex_strcmp:D {#1} {#2} }
5825  \cs_if_exist:NT \luatex_luatexversion:D
5826    {
5827      \cs_set:Npn \__str_if_eq_x:nn #1#2
```

</div>
</div>

392

```
5828            {
5829                \luatex_directlua:D
5830                  {
5831                      l3kernel.strcmp
5832                        (
5833                            " \__str_escape_x:n {#1} " ,
5834                            " \__str_escape_x:n {#2} "
5835                        )
5836                  }
5837            }
5838        \cs_new:Npn \__str_escape_x:n #1
5839            {
5840                \luatex_luaescapestring:D
5841                  {
5842                      \etex_detokenize:D \exp_after:wN { \luatex_expanded:D {#1} }
5843                  }
5844            }
5845      }
```

(*End definition for* __str_if_eq_x:nn *and* __str_escape_x:n.)

__str_if_eq_x_return:nn It turns out that we often need to compare a token list with the result of applying some function to it, and return with \prg_return_true/false:. This test is similar to \str_if_eq:nnTF (see l3str), but is hard-coded for speed.

```
5846 \cs_new:Npn \__str_if_eq_x_return:nn #1 #2
5847    {
5848      \if_int_compare:w \__str_if_eq_x:nn {#1} {#2} = \c_zero
5849        \prg_return_true:
5850      \else:
5851        \prg_return_false:
5852      \fi:
5853    }
```

(*End definition for* __str_if_eq_x_return:nn.)

\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:nn*TF*
\str_if_eq:Vn*TF*
\str_if_eq:on*TF*
\str_if_eq:nV*TF*
\str_if_eq:no*TF*
\str_if_eq:VV*TF*
\str_if_eq_x_p:nn
\str_if_eq_x:nn*TF*

Modern engines provide a direct way of comparing two token lists, but returning a number. This set of conditionals therefore make life a bit clearer. The nn and xx versions are created directly as this is most efficient.

```
5854 \prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
5855    {
5856      \if_int_compare:w
5857        \__str_if_eq_x:nn { \exp_not:n {#1} } { \exp_not:n {#2} }
5858        = \c_zero
5859        \prg_return_true: \else: \prg_return_false: \fi:
5860    }
5861 \cs_generate_variant:Nn \str_if_eq_p:nn {  V ,  o }
5862 \cs_generate_variant:Nn \str_if_eq_p:nn { nV , no , VV }
5863 \cs_generate_variant:Nn \str_if_eq:nnT {  V ,  o }
5864 \cs_generate_variant:Nn \str_if_eq:nnT { nV , no , VV }
5865 \cs_generate_variant:Nn \str_if_eq:nnF {  V ,  o }
5866 \cs_generate_variant:Nn \str_if_eq:nnF { nV , no , VV }
5867 \cs_generate_variant:Nn \str_if_eq:nnTF {  V ,  o }
5868 \cs_generate_variant:Nn \str_if_eq:nnTF { nV , no , VV }
5869 \prg_new_conditional:Npnn \str_if_eq_x:nn #1#2 { p , T , F , TF }
```

393

```
5870    {
5871      \if_int_compare:w \__str_if_eq_x:nn {#1} {#2} = \c_zero
5872        \prg_return_true: \else: \prg_return_false: \fi:
5873    }
```

(*End definition for* `\str_if_eq:nnTF` *and* `\str_if_eq_x:nnTF`*. These functions are documented on page* *104.*)

`\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:NN` is different from `\tl_if_eq:NN` because it needs to ignore category codes.

```
5874 \prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
5875    {
5876      \if_int_compare:w \__str_if_eq_x:nn { \tl_to_str:N #1 } { \tl_to_str:N #2 }
5877        = \c_zero \prg_return_true: \else: \prg_return_false: \fi:
5878    }
5879 \cs_generate_variant:Nn \str_if_eq:NNT  { c , Nc , cc }
5880 \cs_generate_variant:Nn \str_if_eq:NNF  { c , Nc , cc }
5881 \cs_generate_variant:Nn \str_if_eq:NNTF { c , Nc , cc }
5882 \cs_generate_variant:Nn \str_if_eq_p:NN { c , Nc , cc }
```

(*End definition for* `\str_if_eq:NNTF`*. This function is documented on page* *104.*)

`\str_case:nn`
`\str_case:on`
`\str_case:nV`
`\str_case:nv`
`\str_case:nnTF`
`\str_case:onTF`
`\str_case:nVTF`
`\str_case:nvTF`
`\str_case_x:nn`
`\str_case_x:nnTF`
`\__str_case:nnTF`
`\__str_case_x:nnTF`
`\__str_case:nw`
`\__str_case_x:nw`
`\__str_case_end:nw`

Much the same as `\tl_case:nn(TF)` here: just a change in the internal comparison.

```
5883 \cs_new:Npn \str_case:nn #1#2
5884    {
5885      \exp:w
5886      \__str_case:nnTF {#1} {#2} { } { }
5887    }
5888 \cs_new:Npn \str_case:nnT #1#2#3
5889    {
5890      \exp:w
5891      \__str_case:nnTF {#1} {#2} {#3} { }
5892    }
5893 \cs_new:Npn \str_case:nnF #1#2
5894    {
5895      \exp:w
5896      \__str_case:nnTF {#1} {#2} { }
5897    }
5898 \cs_new:Npn \str_case:nnTF #1#2
5899    {
5900      \exp:w
5901      \__str_case:nnTF {#1} {#2}
5902    }
5903 \cs_new:Npn \__str_case:nnTF #1#2#3#4
5904    { \__str_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
5905 \cs_generate_variant:Nn \str_case:nn   { o , nV , nv }
5906 \cs_generate_variant:Nn \str_case:nnT  { o , nV , nv }
5907 \cs_generate_variant:Nn \str_case:nnF  { o , nV , nv }
5908 \cs_generate_variant:Nn \str_case:nnTF { o , nV , nv }
5909 \cs_new:Npn \__str_case:nw #1#2#3
5910    {
5911      \str_if_eq:nnTF {#1} {#2}
5912        { \__str_case_end:nw {#3} }
5913        { \__str_case:nw {#1} }
5914    }
```

```
5915 \cs_new:Npn \str_case_x:nn #1#2
5916   {
5917     \exp:w
5918     \__str_case_x:nnTF {#1} {#2} { } { }
5919   }
5920 \cs_new:Npn \str_case_x:nnT #1#2#3
5921   {
5922     \exp:w
5923     \__str_case_x:nnTF {#1} {#2} {#3} { }
5924   }
5925 \cs_new:Npn \str_case_x:nnF #1#2
5926   {
5927     \exp:w
5928     \__str_case_x:nnTF {#1} {#2} { }
5929   }
5930 \cs_new:Npn \str_case_x:nnTF #1#2
5931   {
5932     \exp:w
5933     \__str_case_x:nnTF {#1} {#2}
5934   }
5935 \cs_new:Npn \__str_case_x:nnTF #1#2#3#4
5936   { \__str_case_x:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
5937 \cs_new:Npn \__str_case_x:nw #1#2#3
5938   {
5939     \str_if_eq_x:nnTF {#1} {#2}
5940       { \__str_case_end:nw {#3} }
5941       { \__str_case_x:nw {#1} }
5942   }
5943 \cs_new_eq:NN \__str_case_end:nw \__prg_case_end:nw
```

(*End definition for* `\str_case:nnTF` *and others. These functions are documented on page* *104.*)

11.3   Accessing specific characters in a string

`\__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 `\q_mark` and `\q_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 `\q_mark` and the first `A` (well, there is also the need to remove a space).

```
5944 \cs_new:Npn \__str_to_other:n #1
5945   {
5946     \exp_after:wN \__str_to_other_loop:w
5947       \tl_to_str:n {#1} ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_mark \q_stop
5948   }
5949 \group_begin:
5950 \tex_lccode:D `\* = `\  %
5951 \tex_lccode:D `\A = `\A
5952 \tex_lowercase:D
5953   {
5954     \group_end:
5955     \cs_new:Npn \__str_to_other_loop:w
5956       #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \q_stop
5957       {
```

```
5958            \if_meaning:w A #8
5959              \__str_to_other_end:w
5960            \fi:
5961            \__str_to_other_loop:w
5962            #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \q_stop
5963          }
5964        \cs_new:Npn \__str_to_other_end:w \fi: #1 \q_mark #2 * A #3 \q_stop
5965          { \fi: #2 }
5966      }
```

(*End definition for* __str_to_other:n *,* __str_to_other_loop:w *, and* __str_to_other_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.

```
5967  \cs_new:Npn \str_item:Nn { \exp_args:No \str_item:nn }
5968  \cs_generate_variant:Nn \str_item:Nn { c }
5969  \cs_new:Npn \str_item:nn #1#2
5970    {
5971      \exp_args:Nf \tl_to_str:n
5972        {
5973          \exp_args:Nf \__str_item:nn
5974            { \__str_to_other:n {#1} } {#2}
5975        }
5976    }
5977  \cs_new:Npn \str_item_ignore_spaces:nn #1
5978    { \exp_args:No \__str_item:nn { \tl_to_str:n {#1} } }
5979  \cs_new:Npn \__str_item:nn #1#2
5980    {
5981      \exp_after:wN \__str_item:w
5982      \__int_value:w \__int_eval:w #2 \exp_after:wN ;
5983      \__int_value:w \__str_count:n {#1} ;
5984      #1 \q_stop
5985    }
5986  \cs_new:Npn \__str_item:w #1; #2;
5987    {
5988      \int_compare:nNnTF {#1} < \c_zero
5989        {
5990          \int_compare:nNnTF {#1} < {-#2}
5991            { \use_none_delimit_by_q_stop:w }
5992            {
5993              \exp_after:wN \use_i_delimit_by_q_stop:nw
5994              \exp:w \exp_after:wN \__str_skip_exp_end:w
5995                \__int_value:w \__int_eval:w #1 + #2 ;
5996            }
```

```
5997            }
5998          {
5999            \int_compare:nNnTF {#1} > {#2}
6000              { \use_none_delimit_by_q_stop:w }
6001              {
6002                \exp_after:wN \use_i_delimit_by_q_stop:nw
6003                \exp:w \__str_skip_exp_end:w #1 ; { }
6004              }
6005          }
6006      }
```

(*End definition for* \str_item:Nn *and others. These functions are documented on page 106.*)

__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).

```
6007  \cs_new:Npn \__str_skip_exp_end:w #1;
6008    {
6009      \if_int_compare:w #1 > \c_eight
6010        \exp_after:wN \__str_skip_loop:wNNNNNNNN
6011      \else:
6012        \exp_after:wN \__str_skip_end:w
6013        \__int_value:w \__int_eval:w
6014      \fi:
6015      #1 ;
6016    }
6017  \cs_new:Npn \__str_skip_loop:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
6018    { \exp_after:wN \__str_skip_exp_end:w \__int_value:w \__int_eval:w #1 - \c_eight ; }
6019  \cs_new:Npn \__str_skip_end:w #1 ;
6020    {
6021      \exp_after:wN \__str_skip_end:NNNNNNNN
6022      \if_case:w #1 \exp_stop_f: \or: \or: \or: \or: \or: \or: \or:
6023    }
6024  \cs_new:Npn \__str_skip_end:NNNNNNNN #1#2#3#4#5#6#7#8 { \fi: \exp_end: }
```

(*End 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.

```
6025  \cs_new:Npn \str_range:Nnn { \exp_args:No \str_range:nnn }
6026  \cs_generate_variant:Nn \str_range:Nnn { c }
6027  \cs_new:Npn \str_range:nnn #1#2#3
6028    {
```

```
6029      \exp_args:Nf \tl_to_str:n
6030        {
6031          \exp_args:Nf \__str_range:nnn
6032            { \__str_to_other:n {#1} } {#2} {#3}
6033        }
6034    }
6035  \cs_new:Npn \str_range_ignore_spaces:nnn #1
6036    { \exp_args:No \__str_range:nnn { \tl_to_str:n {#1} } }
6037  \cs_new:Npn \__str_range:nnn #1#2#3
6038    {
6039      \exp_after:wN \__str_range:w
6040      \__int_value:w \__str_count:n {#1} \exp_after:wN ;
6041      \__int_value:w \__int_eval:w #2 - \c_one \exp_after:wN ;
6042      \__int_value:w \__int_eval:w #3 ;
6043      #1 \q_stop
6044    }
6045  \cs_new:Npn \__str_range:w #1; #2; #3;
6046    {
6047      \exp_args:Nf \__str_range:nnw
6048        { \__str_range_normalize:nn {#2} {#1} }
6049        { \__str_range_normalize:nn {#3} {#1} }
6050    }
6051  \cs_new:Npn \__str_range:nnw #1#2
6052    {
6053      \exp_after:wN \__str_collect_delimit_by_q_stop:w
6054      \__int_value:w \__int_eval:w #2 - #1 \exp_after:wN ;
6055      \exp:w \__str_skip_exp_end:w #1 ;
6056    }
```

(*End definition for* `\str_range:Nnn` *and others. These functions are documented on page 107.*)

`\__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`].

```
6057  \cs_new:Npn \__str_range_normalize:nn #1#2
6058    {
6059      \int_eval:n
6060        {
6061          \if_int_compare:w #1 < \c_zero
6062            \if_int_compare:w #1 < -#2 \exp_stop_f:
6063              \c_zero
6064            \else:
6065              #1 + #2 + \c_one
6066            \fi:
6067          \else:
6068            \if_int_compare:w #1 < #2 \exp_stop_f:
6069              #1
6070            \else:
6071              #2
6072            \fi:
6073          \fi:
6074        }
6075    }
```

(*End definition for* `\__str_range_normalize:nn`.)

Collects max(#1,0) characters, and removes everything else until `\q_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 will close the conditional properly and the `\or:` disappear.

```
6076 \cs_new:Npn \__str_collect_delimit_by_q_stop:w #1;
6077   { \__str_collect_loop:wn #1 ; { } }
6078 \cs_new:Npn \__str_collect_loop:wn #1 ;
6079   {
6080     \if_int_compare:w #1 > \c_seven
6081       \exp_after:wN \__str_collect_loop:wnNNNNNNN
6082     \else:
6083       \exp_after:wN \__str_collect_end:wn
6084     \fi:
6085     #1 ;
6086   }
6087 \cs_new:Npn \__str_collect_loop:wnNNNNNNN #1; #2 #3#4#5#6#7#8#9
6088   {
6089     \exp_after:wN \__str_collect_loop:wn
6090     \__int_value:w \__int_eval:w #1 - \c_seven ;
6091     { #2 #3#4#5#6#7#8#9 }
6092   }
6093 \cs_new:Npn \__str_collect_end:wn #1 ;
6094   {
6095     \exp_after:wN \__str_collect_end:nnnnnnnnw
6096     \if_case:w \if_int_compare:w #1 > \c_zero #1 \else: 0 \fi: \exp_stop_f:
6097     \or: \or: \or: \or: \or: \or: \fi:
6098   }
6099 \cs_new:Npn \__str_collect_end:nnnnnnnnw #1#2#3#4#5#6#7#8 #9 \q_stop
6100   { #1#2#3#4#5#6#7#8 }
```

(*End definition for* `\__str_collect_delimit_by_q_stop:w` *and others.*)

11.4 Counting characters

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$\langle number \rangle$, and that $\langle number \rangle$ is added to the sum of 9 that precedes, to adjust the result.

```
6101 \cs_new:Npn \str_count_spaces:N
6102   { \exp_args:No \str_count_spaces:n }
6103 \cs_generate_variant:Nn \str_count_spaces:N { c }
6104 \cs_new:Npn \str_count_spaces:n #1
6105   {
6106     \int_eval:n
6107       {
6108         \exp_after:wN \__str_count_spaces_loop:w
6109         \tl_to_str:n {#1} ~
6110         X 7 ~ X 6 ~ X 5 ~ X 4 ~ X 3 ~ X 2 ~ X 1 ~ X 0 ~ X -1 ~
6111         \q_stop
```

```
6112          }
6113      }
6114 \cs_new:Npn \__str_count_spaces_loop:w #1~#2~#3~#4~#5~#6~#7~#8~#9~
6115      {
6116        \if_meaning:w X #9
6117          \use_i_delimit_by_q_stop:nw
6118        \fi:
6119        \c_nine + \__str_count_spaces_loop:w
6120      }
```

(*End definition for* \str_count_spaces:N, \str_count_spaces:n, *and* __str_count_spaces_loop:w. *These functions are documented on page 105.*)

\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 __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.

```
6121 \cs_new:Npn \str_count:N { \exp_args:No \str_count:n }
6122 \cs_generate_variant:Nn \str_count:N { c }
6123 \cs_new:Npn \str_count:n #1
6124      {
6125        \__str_count_aux:n
6126          {
6127            \str_count_spaces:n {#1}
6128            + \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1}
6129          }
6130      }
6131 \cs_new:Npn \__str_count:n #1
6132      {
6133        \__str_count_aux:n
6134          { \__str_count_loop:NNNNNNNNN #1 }
6135      }
6136 \cs_new:Npn \str_count_ignore_spaces:n #1
6137      {
6138        \__str_count_aux:n
6139          { \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1} }
6140      }
6141 \cs_new:Npn \__str_count_aux:n #1
6142      {
6143        \int_eval:n
6144          {
6145            #1
6146            { X \c_eight } { X \c_seven } { X \c_six   }
6147            { X \c_five  } { X \c_four  } { X \c_three }
6148            { X \c_two   } { X \c_one   } { X \c_zero  }
6149            \q_stop
6150          }
6151      }
```

```
6152  \cs_new:Npn \__str_count_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
6153    {
6154      \if_meaning:w X #9
6155        \exp_after:wN \use_none_delimit_by_q_stop:w
6156      \fi:
6157      \c_nine + \__str_count_loop:NNNNNNNNN
6158    }
```

(*End definition for* `\str_count:N` *and others. These functions are documented on page 105.*)

11.5 The first character in a string

<div style="text-align:right">

`\str_head:N`
`\str_head:c`
`\str_head:n`
`\str_head_ignore_spaces:n`
`\__str_head:w`

</div>

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, `\use_i_delimit_by_q_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 `\use_i_-delimit_by_q_stop:nw`.

```
6159  \cs_new:Npn \str_head:N { \exp_args:No \str_head:n }
6160  \cs_generate_variant:Nn \str_head:N { c }
6161  \cs_set:Npn \str_head:n #1
6162    {
6163      \exp_after:wN \__str_head:w
6164      \tl_to_str:n {#1}
6165      { { } } ~ \q_stop
6166    }
6167  \cs_set:Npn \__str_head:w #1 ~ %
6168    { \use_i_delimit_by_q_stop:nw #1 { ~ } }
6169  \cs_new:Npn \str_head_ignore_spaces:n #1
6170    {
6171      \exp_after:wN \use_i_delimit_by_q_stop:nw
6172      \tl_to_str:n {#1} { } \q_stop
6173    }
```

(*End definition for* `\str_head:N` *and others. These functions are documented on page 106.*)

<div style="text-align:right">

`\str_tail:N`
`\str_tail:c`
`\str_tail:n`
`\str_tail_ignore_spaces:n`
`\__str_tail_auxi:w`
`\__str_tail_auxii:w`

</div>

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 `\q_mark`. One can check that an empty (or blank) string yields an empty tail.

```
6174  \cs_new:Npn \str_tail:N { \exp_args:No \str_tail:n }
6175  \cs_generate_variant:Nn \str_tail:N { c }
6176  \cs_set:Npn \str_tail:n #1
```

```
6177    {
6178      \exp_after:wN \__str_tail_auxi:w
6179      \reverse_if:N \if_charcode:w
6180        \scan_stop: \tl_to_str:n {#1} X X \q_stop
6181    }
6182  \cs_set:Npn \__str_tail_auxi:w #1 X #2 \q_stop { \fi: #1 }
6183  \cs_new:Npn \str_tail_ignore_spaces:n #1
6184    {
6185      \exp_after:wN \__str_tail_auxii:w
6186      \tl_to_str:n {#1} \q_mark \q_mark \q_stop
6187    }
6188  \cs_new:Npn \__str_tail_auxii:w #1 #2 \q_mark #3 \q_stop { #2 }
```

(*End definition for* \str_tail:N *and others. These functions are documented on page 106.*)

11.6 String manipulation

\str_fold_case:n
\str_fold_case:V
\str_lower_case:n
\str_lower_case:f
\str_upper_case:n
\str_upper_case: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_lookup_lower:N
__str_lookup_upper:N
__str_lookup_fold:N

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.

```
6189  \cs_new:Npn \str_fold_case:n  #1 { \__str_change_case:nn {#1} { fold } }
6190  \cs_new:Npn \str_lower_case:n #1 { \__str_change_case:nn {#1} { lower } }
6191  \cs_new:Npn \str_upper_case:n #1 { \__str_change_case:nn {#1} { upper } }
6192  \cs_generate_variant:Nn \str_fold_case:n  { V }
6193  \cs_generate_variant:Nn \str_lower_case:n { f }
6194  \cs_generate_variant:Nn \str_upper_case:n { f }
6195  \cs_new:Npn \__str_change_case:nn #1
6196    {
6197      \exp_after:wN \__str_change_case_aux:nn \exp_after:wN
6198        { \tl_to_str:n {#1} }
6199    }
6200  \cs_new:Npn \__str_change_case_aux:nn #1#2
6201    {
6202      \__str_change_case_loop:nw {#2} #1 \q_recursion_tail \q_recursion_stop
6203        \__str_change_case_result:n { }
6204    }
6205  \cs_new:Npn \__str_change_case_output:nw #1#2 \__str_change_case_result:n #3
6206    { #2 \__str_change_case_result:n { #3 #1 } }
6207  \cs_generate_variant:Nn  \__str_change_case_output:nw { f }
6208  \cs_new:Npn \__str_change_case_end:wn #1 \__str_change_case_result:n #2 { #2 }
6209  \cs_new:Npn \__str_change_case_loop:nw #1#2 \q_recursion_stop
6210    {
6211      \tl_if_head_is_space:nTF {#2}
6212        { \__str_change_case_space:n }
6213        { \__str_change_case_char:nN }
6214      {#1} #2 \q_recursion_stop
6215    }
6216  \use:x
6217    { \cs_new:Npn \exp_not:N \__str_change_case_space:n ##1 \c_space_tl }
6218    {
6219      \__str_change_case_output:nw { ~ }
6220      \__str_change_case_loop:nw {#1}
6221    }
6222  \cs_new:Npn \__str_change_case_char:nN #1#2
```

```
6223    {
6224      \quark_if_recursion_tail_stop_do:Nn #2
6225        { \__str_change_case_end:wn }
6226      \cs_if_exist:cTF { c__unicode_ #1 _ #2 _tl }
6227        {
6228          \__str_change_case_output:fw
6229            { \tl_to_str:c { c__unicode_ #1 _ #2 _tl } }
6230        }
6231        { \__str_change_case_char_aux:nN {#1} #2 }
6232      \__str_change_case_loop:nw {#1}
6233    }
```

For Unicode engines there's a look up to see if the current character has a valid one-to-one case change mapping. That's not needed for 8-bit engines: as they don't have `\utex_char:D` all of the changes they can make are hard-coded and so already picked up above.

```
6234 \cs_if_exist:NTF \utex_char:D
6235    {
6236      \cs_new:Npn \__str_change_case_char_aux:nN #1#2
6237        {
6238          \int_compare:nNnTF { \use:c { __str_lookup_ #1 :N } } #2 = { 0 }
6239            { \__str_change_case_output:nw {#2} }
6240            {
6241              \__str_change_case_output:fw
6242                { \utex_char:D \use:c { __str_lookup_ #1 :N } } #2 ~ }
6243            }
6244        }
6245      \cs_set_protected:Npn \__str_lookup_lower:N #1 { \tex_lccode:D `#1 }
6246      \cs_set_protected:Npn \__str_lookup_upper:N #1 { \tex_uccode:D `#1 }
6247      \cs_set_eq:NN \__str_lookup_fold:N \__str_lookup_lower:N
6248    }
6249    {
6250      \cs_new:Npn \__str_change_case_char_aux:nN #1#2
6251        { \__str_change_case_output:nw {#2} }
6252    }
```

(*End definition for* `\str_fold_case:n` *and others. These functions are documented on page 109.*)

`\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`

For all of those strings, use `\cs_to_str:N` to get characters with the correct category code without worries

```
6253 \str_const:Nx \c_ampersand_str    { \cs_to_str:N \& }
6254 \str_const:Nx \c_atsign_str       { \cs_to_str:N \@ }
6255 \str_const:Nx \c_backslash_str    { \cs_to_str:N \\ }
6256 \str_const:Nx \c_left_brace_str   { \cs_to_str:N \{ }
6257 \str_const:Nx \c_right_brace_str  { \cs_to_str:N \} }
6258 \str_const:Nx \c_circumflex_str   { \cs_to_str:N \^ }
6259 \str_const:Nx \c_colon_str        { \cs_to_str:N \: }
6260 \str_const:Nx \c_dollar_str       { \cs_to_str:N \$ }
6261 \str_const:Nx \c_hash_str         { \cs_to_str:N \# }
6262 \str_const:Nx \c_percent_str      { \cs_to_str:N \% }
6263 \str_const:Nx \c_tilde_str        { \cs_to_str:N \~ }
6264 \str_const:Nx \c_underscore_str   { \cs_to_str:N \_ }
```

(*End definition for* `\c_ampersand_str` *and others. These variables are documented on page 110.*)

`\l_tmpa_str`
`\l_tmpb_str`
`\g_tmpa_str`
`\g_tmpb_str`

Scratch strings.

```
6265 \str_new:N \l_tmpa_str
6266 \str_new:N \l_tmpb_str
6267 \str_new:N \g_tmpa_str
6268 \str_new:N \g_tmpb_str
```

(*End definition for* `\l_tmpa_str` *and others. These variables are documented on page 110.*)

11.7 Viewing strings

`\str_show:n`
`\str_show:N`
`\str_show:c`

Displays a string on the terminal.

```
6269 \cs_new_eq:NN \str_show:n \tl_show:n
6270 \cs_new_eq:NN \str_show:N \tl_show:N
6271 \cs_generate_variant:Nn \str_show:N { c }
```

(*End definition for* `\str_show:n` *and* `\str_show:N`. *These functions are documented on page 109.*)

11.8 Unicode data for case changing

```
6272 ⟨@@=unicode⟩
```

Case changing both for strings and "text" requires data from the Unicode Consortium. Some of this is build in to the format (as `\lccode` and `\uccode` values) but this covers only the simple one-to-one situations and does not fully handle for example case folding.

The data required for cross-module manipulations is loaded here: currently this means for `str` and `tl` functions. As such, the prefix used is not `str` but rather `unicode`. For performance (as the entire data set must be read during each run) and as this code comes somewhat early in the load process, there is quite a bit of low-level code here.

As only the data needs to remain at the end of this process, everything is set up inside a group.

```
6273 \group_begin:
```

A read stream is needed. The I/O module is not yet in place *and* we do not want to use up a stream. We therefore use a known free one in format mode or look for the next free one in package mode (covers plain, LATEX 2ε and ConTEXt MkII and MkIV).

```
6274 ⟨*initex⟩
6275     \tex_chardef:D \g__unicode_data_ior \c_zero
6276 ⟨/initex⟩
6277 ⟨*package⟩
6278     \tex_chardef:D \g__unicode_data_ior
6279       \etex_numexpr:D
6280         \cs_if_exist:NTF \lastallocatedread
6281           { \lastallocatedread }
6282           {
6283             \cs_if_exist:NTF \c_syst_last_allocated_read
6284               { \c_syst_last_allocated_read }
6285               { \tex_count:D 16 ~ }
6286           }
6287         + 1
6288     \scan_stop:
6289 ⟨/package⟩
```

Set up to read each file. As they use C-style comments, there is a need to deal with #. At the same time, spaces are important so they need to be picked up as they are important.

404

Beyond that, the current category code scheme works fine. With no I/O loop available, hard-code one that will work quickly.

```
6290    \cs_set_protected:Npn \__unicode_map_inline:n #1
6291      {
6292        \group_begin:
6293          \tex_catcode:D `\# = 12 \scan_stop:
6294          \tex_catcode:D `\  = 10 \scan_stop:
6295          \tex_openin:D \g__unicode_data_ior = #1 \scan_stop:
6296          \cs_if_exist:NT \utex_char:D
6297            { \__unicode_map_loop: }
6298          \tex_closein:D \g__unicode_data_ior
6299        \group_end:
6300      }
6301    \cs_set_protected:Npn \__unicode_map_loop:
6302      {
6303        \tex_ifeof:D \g__unicode_data_ior
6304          \exp_after:wN \use_none:n
6305        \else:
6306          \exp_after:wN \use:n
6307        \fi:
6308          {
6309            \tex_read:D \g__unicode_data_ior to \l__unicode_tmp_tl
6310            \if_meaning:w \c_empty_tl \l__unicode_tmp_tl
6311            \else:
6312              \exp_after:wN \__unicode_parse:w \l__unicode_tmp_tl \q_stop
6313            \fi:
6314            \__unicode_map_loop:
6315          }
6316      }
```

The lead-off parser for each line is common for all of the files. If the line starts with a `#` it's a comment. There's one special comment line to look out for in `SpecialCasing.txt` as we want to ignore everything after it. As this line does not appear in any other sources and the test is quite quick (there are relatively few comment lines), it can be present in all of the passes.

```
6317    \cs_set_protected:Npn \__unicode_parse:w #1#2 \q_stop
6318      {
6319        \reverse_if:N \if:w \c_hash_str #1
6320          \__unicode_parse_auxi:w #1#2 \q_stop
6321        \else:
6322          \if_int_compare:w \__str_if_eq_x:nn
6323            { \exp_not:n {#2} } { ~Conditional~Mappings~ } = \c_zero
6324            \cs_set_protected:Npn \__unicode_parse:w ##1 \q_stop { }
6325          \fi:
6326        \fi:
6327      }
```

Storing each exception is always done in the same way: create a constant token list which expands to exactly the mapping. These will have the category codes "now" (so should be letters) but will be detokenized for string use.

```
6328    \cs_set_protected:Npn \__unicode_store:nnnnn #1#2#3#4#5
6329      {
6330        \tl_const:cx { c__unicode_ #2 _ \utex_char:D "#1 _tl }
6331          {
6332            \utex_char:D "#3 ~
```

```
6333          \utex_char:D "#4 ~
6334          \tl_if_blank:nF {#5}
6335            { \utex_char:D "#5 }
6336        }
6337      }
```

Parse the main Unicode data file for title case exceptions (the one-to-one lower and upper case mappings it contains will all be covered by the TeX data).

```
6338    \cs_set_protected:Npn \__unicode_parse_auxi:w
6339      #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ;
6340      { \__unicode_parse_auxii:w #1 ; }
6341    \cs_set_protected:Npn \__unicode_parse_auxii:w
6342      #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 \q_stop
6343      {
6344        \tl_if_blank:nF {#7}
6345          {
6346            \if_int_compare:w \__str_if_eq_x:nn { #5 ~ } {#7} = \c_zero
6347            \else:
6348              \tl_const:cx
6349                { c__unicode_title_ \utex_char:D "#1 _tl }
6350                { \utex_char:D "#7 }
6351            \fi:
6352          }
6353      }
6354    \__unicode_map_inline:n { UnicodeData.txt }
```

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.

```
6355    \cs_set_protected:Npn \__unicode_parse_auxi:w #1 ;~ #2 ;~ #3 ; #4 \q_stop
6356      {
6357        \if_int_compare:w \__str_if_eq_x:nn {#2} { C } = \c_zero
6358          \if_int_compare:w \tex_lccode:D "#1 = "#3 \scan_stop:
6359          \else:
6360            \tl_const:cx
6361              { c__unicode_fold_ \utex_char:D "#1 _tl }
6362              { \utex_char:D "#3 ~ }
6363          \fi:
6364        \else:
6365          \if_int_compare:w \__str_if_eq_x:nn {#2} { F } = \c_zero
6366            \__unicode_parse_auxii:w #1 ~ #3 ~ \q_stop
6367          \fi:
6368        \fi:
6369      }
6370    \cs_set_protected:Npn \__unicode_parse_auxii:w #1 ~ #2 ~ #3 ~ #4 \q_stop
6371      { \__unicode_store:nnnnn {#1} { fold } {#2} {#3} {#4} }
6372    \__unicode_map_inline:n { CaseFolding.txt }
```

For upper and lower casing special situations, there is a bit more to do as we also have title casing to consider.

```
6373    \cs_set_protected:Npn \__unicode_parse_auxi:w #1 ;~ #2 ;~ #3 ;~ #4 ; #5 \q_stop
6374      {
6375        \use:n { \__unicode_parse_auxii:w #1 ~ lower ~ #2 ~ } ~ \q_stop
6376        \use:n { \__unicode_parse_auxii:w #1 ~ upper ~ #4 ~ } ~ \q_stop
6377        \if_int_compare:w \__str_if_eq_x:nn {#3} {#4} = \c_zero
```

```
6378        \else:
6379          \use:n { \__unicode_parse_auxii:w #1 ~ title ~ #3 ~ } ~ \q_stop
6380        \fi:
6381      }
6382    \cs_set_protected:Npn \__unicode_parse_auxii:w #1 ~ #2 ~ #3 ~ #4 ~ #5 \q_stop
6383      {
6384        \tl_if_empty:nF {#4}
6385          { \__unicode_store:nnnnn {#1} {#2} {#3} {#4} {#5} }
6386      }
6387    \__unicode_map_inline:n { SpecialCasing.txt }
```

For the 8-bit engines, the above does nothing but there is some set up needed. There is no expandable character generator primitive so some alternative is needed. As we've not used up hash space for the above, we can go for the fast approach here of one name per letter. Keeping folding and lower casing separate makes the use later a bit easier.

```
6388    \cs_if_exist:NF \utex_char:D
6389      {
6390        \cs_set_protected:Npn \__unicode_tmp:NN #1#2
6391          {
6392            \if_meaning:w \q_recursion_tail #2
6393              \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
6394            \fi:
6395            \tl_const:cn { c__unicode_fold_  #1 _tl } {#2}
6396            \tl_const:cn { c__unicode_lower_ #1 _tl } {#2}
6397            \tl_const:cn { c__unicode_upper_ #2 _tl } {#1}
6398            \__unicode_tmp:NN
6399          }
6400        \__unicode_tmp:NN
6401          AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
6402          ? \q_recursion_tail \q_recursion_stop
6403      }
```

All done: tidy up.

```
6404  \group_end:
6405  ⟨/initex | package⟩
```

12   l3seq implementation

The following test files are used for this code: m3seq002,m3seq003.

```
6406  ⟨*initex | package⟩
```

```
6407  ⟨@@=seq⟩
```

A sequence is a control sequence whose top-level expansion is of the form "\s__-seq __seq_item:n {⟨item₁⟩} ... __seq_item:n {⟨itemₙ⟩}", with a leading scan mark followed by n items of the same form. An earlier implementation used the structure "\seq_elt:w ⟨item₁⟩ \seq_elt_end: ...\seq_elt:w ⟨itemₙ⟩ \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.

\s__seq   The variable is defined in the l3quark module, loaded later.

(*End definition for* \s__seq.)

__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.

```
6408 \cs_new:Npn \__seq_item:n
6409   {
6410     \__msg_kernel_expandable_error:nn { kernel } { misused-sequence }
6411     \use_none:n
6412   }
```

(*End definition for* __seq_item:n.)

\l__seq_internal_a_tl Scratch space for various internal uses.
\l__seq_internal_b_tl
```
6413 \tl_new:N \l__seq_internal_a_tl
6414 \tl_new:N \l__seq_internal_b_tl
```

(*End definition for* \l__seq_internal_a_tl *and* \l__seq_internal_b_tl.)

__seq_tmp:w Scratch function for internal use.
```
6415 \cs_new_eq:NN \__seq_tmp:w ?
```

(*End definition for* __seq_tmp:w.)

\c_empty_seq A sequence with no item, following the structure mentioned above.
```
6416 \tl_const:Nn \c_empty_seq { \s__seq }
```

(*End definition for* \c_empty_seq. *This variable is documented on page 121.*)

12.1 Allocation and initialisation

\seq_new:N Sequences are initialized to \c_empty_seq.
\seq_new:c
```
6417 \cs_new_protected:Npn \seq_new:N #1
6418   {
6419     \__chk_if_free_cs:N #1
6420     \cs_gset_eq:NN #1 \c_empty_seq
6421   }
6422 \cs_generate_variant:Nn \seq_new:N { c }
```

(*End definition for* \seq_new:N. *This function is documented on page 112.*)

\seq_clear:N Clearing a sequence is similar to setting it equal to the empty one.
\seq_clear:c
\seq_gclear:N
\seq_gclear:c
```
6423 \cs_new_protected:Npn \seq_clear:N  #1
6424   { \seq_set_eq:NN #1 \c_empty_seq }
6425 \cs_generate_variant:Nn \seq_clear:N  { c }
6426 \cs_new_protected:Npn \seq_gclear:N #1
6427   { \seq_gset_eq:NN #1 \c_empty_seq }
6428 \cs_generate_variant:Nn \seq_gclear:N { c }
```

(*End definition for* \seq_clear:N *and* \seq_gclear:N. *These functions are documented on page 112.*)

\seq_clear_new:N Once again we copy code from the token list functions.
\seq_clear_new:c
\seq_gclear_new:N
\seq_gclear_new:c
```
6429 \cs_new_protected:Npn \seq_clear_new:N  #1
6430   { \seq_if_exist:NTF #1 { \seq_clear:N #1 } { \seq_new:N #1 } }
6431 \cs_generate_variant:Nn \seq_clear_new:N  { c }
6432 \cs_new_protected:Npn \seq_gclear_new:N #1
6433   { \seq_if_exist:NTF #1 { \seq_gclear:N #1 } { \seq_new:N #1 } }
6434 \cs_generate_variant:Nn \seq_gclear_new:N { c }
```

(*End definition for* `\seq_clear_new:N` *and* `\seq_gclear_new:N`. *These functions are documented on page* *112.*)

`\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.

```
6435 \cs_new_eq:NN \seq_set_eq:NN  \tl_set_eq:NN
6436 \cs_new_eq:NN \seq_set_eq:Nc  \tl_set_eq:Nc
6437 \cs_new_eq:NN \seq_set_eq:cN  \tl_set_eq:cN
6438 \cs_new_eq:NN \seq_set_eq:cc  \tl_set_eq:cc
6439 \cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
6440 \cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
6441 \cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN
6442 \cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc
```

(*End definition for* `\seq_set_eq:NN` *and* `\seq_gset_eq:NN`. *These functions are documented on page* *112.*)

`\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.

```
6443 \cs_new_protected:Npn \seq_set_from_clist:NN #1#2
6444   {
6445     \tl_set:Nx #1
6446       { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
6447   }
6448 \cs_new_protected:Npn \seq_set_from_clist:Nn #1#2
6449   {
6450     \tl_set:Nx #1
6451       { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
6452   }
6453 \cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
6454   {
6455     \tl_gset:Nx #1
6456       { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
6457   }
6458 \cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
6459   {
6460     \tl_gset:Nx #1
6461       { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
6462   }
6463 \cs_generate_variant:Nn \seq_set_from_clist:NN  {     Nc }
6464 \cs_generate_variant:Nn \seq_set_from_clist:NN  { c , cc }
6465 \cs_generate_variant:Nn \seq_set_from_clist:Nn  { c      }
6466 \cs_generate_variant:Nn \seq_gset_from_clist:NN {     Nc }
6467 \cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
6468 \cs_generate_variant:Nn \seq_gset_from_clist:Nn { c      }
```

(*End definition for* `\seq_set_from_clist:NN` *and others. These functions are documented on page* *112.*)

`\seq_set_split:Nnn`
`\seq_set_split:NnV`
`\seq_gset_split:Nnn`
`\seq_gset_split:NnV`
`\__seq_set_split:NNnn`
`\__seq_set_split_auxi:w`
`\__seq_set_split_auxii: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_auxi:w \prg_do_nothing:` ⟨*item with spaces*⟩ `\__seq_set_split_end:`. Then, x-expansion causes `\__seq_set_split_auxi:w` to trim

409

spaces, and leaves its result as `\__seq_set_split_auxii:w` *⟨trimmed item⟩* `\__seq_-set_split_end:`. This is then converted to the l3seq internal structure by another x-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.

```
6469 \cs_new_protected:Npn \seq_set_split:Nnn
6470   { \__seq_set_split:NNnn \tl_set:Nx }
6471 \cs_new_protected:Npn \seq_gset_split:Nnn
6472   { \__seq_set_split:NNnn \tl_gset:Nx }
6473 \cs_new_protected:Npn \__seq_set_split:NNnn #1#2#3#4
6474   {
6475     \tl_if_empty:nTF {#3}
6476       {
6477         \tl_set:Nn \l__seq_internal_a_tl
6478           { \tl_map_function:nN {#4} \__seq_wrap_item:n }
6479       }
6480       {
6481         \tl_set:Nn \l__seq_internal_a_tl
6482           {
6483             \__seq_set_split_auxi:w \prg_do_nothing:
6484             #4
6485             \__seq_set_split_end:
6486           }
6487         \tl_replace_all:Nnn \l__seq_internal_a_tl { #3 }
6488           {
6489             \__seq_set_split_end:
6490             \__seq_set_split_auxi:w \prg_do_nothing:
6491           }
6492         \tl_set:Nx \l__seq_internal_a_tl { \l__seq_internal_a_tl }
6493       }
6494     #1 #2 { \s__seq \l__seq_internal_a_tl }
6495   }
6496 \cs_new:Npn \__seq_set_split_auxi:w #1 \__seq_set_split_end:
6497   {
6498     \exp_not:N \__seq_set_split_auxii:w
6499     \exp_args:No \tl_trim_spaces:n {#1}
6500     \exp_not:N \__seq_set_split_end:
6501   }
6502 \cs_new:Npn \__seq_set_split_auxii:w #1 \__seq_set_split_end:
6503   { \__seq_wrap_item:n {#1} }
6504 \cs_generate_variant:Nn \seq_set_split:Nnn  { NnV }
6505 \cs_generate_variant:Nn \seq_gset_split:Nnn { NnV }
```

(*End definition for* `\seq_set_split:Nnn` *and others. These functions are documented on page 113.*)

`\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.

```
6506 \cs_new_protected:Npn \seq_concat:NNN #1#2#3
6507   { \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
6508 \cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
6509   { \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
6510 \cs_generate_variant:Nn \seq_concat:NNN  { ccc }
6511 \cs_generate_variant:Nn \seq_gconcat:NNN { ccc }
```

(*End definition for* \seq_concat:NNN *and* \seq_gconcat:NNN*. These functions are documented on page [113](#).*)

<div class="marginpar">

\seq_if_exist_p:N
\seq_if_exist_p:c
\seq_if_exist:N*TF*
\seq_if_exist:c*TF*

</div>

Copies of the `cs` functions defined in l3basics.

```
6512 \prg_new_eq_conditional:NNn \seq_if_exist:N \cs_if_exist:N
6513   { TF , T , F , p }
6514 \prg_new_eq_conditional:NNn \seq_if_exist:c \cs_if_exist:c
6515   { TF , T , F , p }
```

(*End definition for* \seq_if_exist:NTF*. This function is documented on page [113](#).*)

12.2 Appending data to either end

<div class="marginpar">

\seq_put_left:Nn
\seq_put_left:NV
\seq_put_left:Nv
\seq_put_left:No
\seq_put_left:Nx
\seq_put_left:cn
\seq_put_left:cV
\seq_put_left:cv
\seq_put_left:co
\seq_put_left:cx
\seq_gput_left:Nn
\seq_gput_left:NV
\seq_gput_left:Nv
\seq_gput_left:No
\seq_gput_left:Nx
\seq_gput_left:cn
\seq_gput_left:cV
\seq_gput_left:cv
\seq_gput_left:co
\seq_gput_left:cx
__seq_put_left_aux:w

</div>

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.

```
6516 \cs_new_protected:Npn \seq_put_left:Nn #1#2
6517   {
6518     \tl_set:Nx #1
6519       {
6520         \exp_not:n { \s__seq \__seq_item:n {#2} }
6521         \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
6522       }
6523   }
6524 \cs_new_protected:Npn \seq_gput_left:Nn #1#2
6525   {
6526     \tl_gset:Nx #1
6527       {
6528         \exp_not:n { \s__seq \__seq_item:n {#2} }
6529         \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
6530       }
6531   }
6532 \cs_new:Npn \__seq_put_left_aux:w \s__seq { \exp_stop_f: }
6533 \cs_generate_variant:Nn \seq_put_left:Nn {    NV , Nv , No , Nx }
6534 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , co , cx }
6535 \cs_generate_variant:Nn \seq_gput_left:Nn {   NV , Nv , No , Nx }
6536 \cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , co , cx }
```

(*End definition for* \seq_put_left:Nn*,* \seq_gput_left:Nn*, and* __seq_put_left_aux:w*. These functions are documented on page [113](#).*)

<div class="marginpar">

\seq_put_right:Nn
\seq_put_right:NV
\seq_put_right:Nv
\seq_put_right:No
\seq_put_right:Nx
\seq_put_right:cn
\seq_put_right:cV
\seq_put_right:cv
\seq_put_right:co
\seq_put_right:cx
\seq_gput_right:Nn
\seq_gput_right:NV
\seq_gput_right:Nv
\seq_gput_right:No
\seq_gput_right:Nx
\seq_gput_right:cn
\seq_gput_right:cV
\seq_gput_right:cv
\seq_gput_right:co
\seq_gput_right:cx

</div>

Since there is no trailing marker, adding an item to the right of a sequence simply means
wrapping it in __seq_item:n.

```
6537 \cs_new_protected:Npn \seq_put_right:Nn #1#2
6538   { \tl_put_right:Nn #1 { \__seq_item:n {#2} } }
6539 \cs_new_protected:Npn \seq_gput_right:Nn #1#2
6540   { \tl_gput_right:Nn #1 { \__seq_item:n {#2} } }
6541 \cs_generate_variant:Nn \seq_gput_right:Nn {    NV , Nv , No , Nx }
6542 \cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , co , cx }
6543 \cs_generate_variant:Nn \seq_put_right:Nn {    NV , Nv , No , Nx }
6544 \cs_generate_variant:Nn \seq_put_right:Nn { c , cV , cv , co , cx }
```

(*End definition for* \seq_put_right:Nn *and* \seq_gput_right:Nn*. These functions are documented on page [113](#).*)

411

12.3 Modifying sequences

__seq_wrap_item:n   This function converts its argument to a proper sequence item in an x-expansion context.

```
6545 \cs_new:Npn \__seq_wrap_item:n #1 { \exp_not:n { \__seq_item:n {#1} } }
```

(*End definition for* __seq_wrap_item:n.)

\l__seq_remove_seq   An internal sequence for the removal routines.

```
6546 \seq_new:N \l__seq_remove_seq
```

(*End 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.

```
6547 \cs_new_protected:Npn \seq_remove_duplicates:N
6548   { \__seq_remove_duplicates:NN \seq_set_eq:NN }
6549 \cs_new_protected:Npn \seq_gremove_duplicates:N
6550   { \__seq_remove_duplicates:NN \seq_gset_eq:NN }
6551 \cs_new_protected:Npn \__seq_remove_duplicates:NN #1#2
6552   {
6553     \seq_clear:N \l__seq_remove_seq
6554     \seq_map_inline:Nn #2
6555       {
6556         \seq_if_in:NnF \l__seq_remove_seq {##1}
6557           { \seq_put_right:Nn \l__seq_remove_seq {##1} }
6558       }
6559     #1 #2 \l__seq_remove_seq
6560   }
6561 \cs_generate_variant:Nn \seq_remove_duplicates:N  { c }
6562 \cs_generate_variant:Nn \seq_gremove_duplicates:N { c }
```

(*End definition for* \seq_remove_duplicates:N, \seq_gremove_duplicates:N, *and* __seq_remove_-
duplicates:NN. *These functions are documented on page* *116.*)

\seq_remove_all:Nn
\seq_remove_all:cn
\seq_gremove_all:Nn
\seq_gremove_all:cn
__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" x-type expansion to do most of the work. As \tl_if_eq:nnT is not expandable, a two-part strategy is needed. First, the x-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 x-type is started again, including all of the items copied already. This will happen repeatedly until the entire sequence has been scanned. The code is set up to avoid needing and intermediate scratch list: the lead-off x-type expansion (#1 #2 {#2}) will ensure that nothing is lost.

```
6563 \cs_new_protected:Npn \seq_remove_all:Nn
6564   { \__seq_remove_all_aux:NNn \tl_set:Nx }
6565 \cs_new_protected:Npn \seq_gremove_all:Nn
6566   { \__seq_remove_all_aux:NNn \tl_gset:Nx }
6567 \cs_new_protected:Npn \__seq_remove_all_aux:NNn #1#2#3
6568   {
6569     \__seq_push_item_def:n
6570       {
6571         \str_if_eq:nnT {##1} {#3}
6572           {
6573             \if_false: { \fi: }
```

```
6574                  \tl_set:Nn \l__seq_internal_b_tl {##1}
6575              #1 #2
6576                { \if_false: } \fi:
6577                  \exp_not:o {#2}
6578                  \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
6579                    { \use_none:nn }
6580            }
6581          \__seq_wrap_item:n {##1}
6582        }
6583      \tl_set:Nn \l__seq_internal_a_tl {#3}
6584      #1 #2 {#2}
6585      \__seq_pop_item_def:
6586    }
6587 \cs_generate_variant:Nn \seq_remove_all:Nn  { c }
6588 \cs_generate_variant:Nn \seq_gremove_all:Nn { c }
```

(*End definition for* \seq_remove_all:Nn, \seq_gremove_all:Nn, *and* __seq_remove_all_aux:NNn.
These functions are documented on page 116.)

\seq_reverse:N   Previously, \seq_reverse:N was coded by collecting the items in reverse order after an
\seq_reverse:c   \exp_stop_f: marker.
\seq_greverse:N
\seq_greverse:c
__seq_reverse:NN
__seq_reverse_item:nwn

```
\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.

```
6589 \cs_new_protected:Npn \seq_reverse:N
6590    { \__seq_reverse:NN \tl_set:Nx }
6591 \cs_new_protected:Npn \seq_greverse:N
6592    { \__seq_reverse:NN \tl_gset:Nx }
6593 \cs_new_protected:Npn \__seq_reverse:NN #1 #2
6594    {
6595      \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
6596      \cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn
6597      #1 #2 { #2 \exp_not:n { } }
```

413

```
6598        \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
6599      }
6600 \cs_new:Npn \__seq_reverse_item:nwn #1 #2 \exp_not:n #3
6601    {
6602      #2
6603      \exp_not:n { \__seq_item:n {#1} #3 }
6604    }
6605 \cs_generate_variant:Nn \seq_reverse:N  { c }
6606 \cs_generate_variant:Nn \seq_greverse:N { c }
```

(*End definition for* `\seq_reverse:N` *and others. These functions are documented on page 116.*)

12.4   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.

```
6607 \prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }
6608    {
6609      \if_meaning:w #1 \c_empty_seq
6610        \prg_return_true:
6611      \else:
6612        \prg_return_false:
6613      \fi:
6614    }
6615 \cs_generate_variant:Nn \seq_if_empty_p:N { c }
6616 \cs_generate_variant:Nn \seq_if_empty:NT { c }
6617 \cs_generate_variant:Nn \seq_if_empty:NF { c }
6618 \cs_generate_variant:Nn \seq_if_empty:NTF { c }
```

(*End definition for* `\seq_if_empty:NTF`. *This function is documented on page 116.*)

\seq_if_in:NnTF
\seq_if_in:NVTF
\seq_if_in:NvTF
\seq_if_in:NoTF
\seq_if_in:NxTF
\seq_if_in:cnTF
\seq_if_in:cVTF
\seq_if_in:cvTF
\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 will break returning `\prg_return_false:`. Everything is inside a group so that `\__seq_item:n` is preserved in nested situations.

```
6619 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
6620    { T , F , TF }
6621    {
6622      \group_begin:
6623        \tl_set:Nn \l__seq_internal_a_tl {#2}
6624        \cs_set_protected:Npn \__seq_item:n ##1
6625          {
6626            \tl_set:Nn \l__seq_internal_b_tl {##1}
6627            \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
6628              \exp_after:wN \__seq_if_in:
6629            \fi:
6630          }
6631        #1
6632      \group_end:
6633      \prg_return_false:
6634      \__prg_break_point:
6635    }
6636 \cs_new:Npn \__seq_if_in:
6637    { \__prg_break:n { \group_end: \prg_return_true: } }
```

414

```
6638 \cs_generate_variant:Nn \seq_if_in:NnT {    NV , Nv , No , Nx }
6639 \cs_generate_variant:Nn \seq_if_in:NnT { c , cV , cv , co , cx }
6640 \cs_generate_variant:Nn \seq_if_in:NnF {    NV , Nv , No , Nx }
6641 \cs_generate_variant:Nn \seq_if_in:NnF { c , cV , cv , co , cx }
6642 \cs_generate_variant:Nn \seq_if_in:NnTF {    NV , Nv , No , Nx }
6643 \cs_generate_variant:Nn \seq_if_in:NnTF { c , cV , cv , co , cx }
```

(*End definition for* \seq_if_in:NnTF *and* __seq_if_in:*. These functions are documented on page* *116.*)

12.5   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.

```
6644 \cs_new_protected:Npn \__seq_pop:NNNN #1#2#3#4
6645   {
6646     \if_meaning:w #3 \c_empty_seq
6647       \tl_set:Nn #4 { \q_no_value }
6648     \else:
6649       #1#2#3#4
6650     \fi:
6651   }
6652 \cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
6653   {
6654     \if_meaning:w #3 \c_empty_seq
6655       % \tl_set:Nn #4 { \q_no_value }
6656       \prg_return_false:
6657     \else:
6658       #1#2#3#4
6659       \prg_return_true:
6660     \fi:
6661   }
```

(*End 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

```
6662 \cs_new_protected:Npn \seq_get_left:NN #1#2
6663   {
6664     \tl_set:Nx #2
6665       {
6666         \exp_after:wN \__seq_get_left:wnw
6667         #1 \__seq_item:n { \q_no_value } \q_stop
6668       }
6669   }
6670 \cs_new:Npn \__seq_get_left:wnw #1 \__seq_item:n #2#3 \q_stop
6671   { \exp_not:n {#2} }
6672 \cs_generate_variant:Nn \seq_get_left:NN { c }
```

(*End definition for* \seq_get_left:NN *and* __seq_get_left:wnw. *These functions are documented on page* *113.*)

415

\seq_pop_left:NN
\seq_pop_left:cN
\seq_gpop_left:NN
\seq_gpop_left:cN
__seq_pop_left:NNNN
__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.

```
6673 \cs_new_protected:Npn \seq_pop_left:NN
6674   { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_set:Nn }
6675 \cs_new_protected:Npn \seq_gpop_left:NN
6676   { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_gset:Nn }
6677 \cs_new_protected:Npn \__seq_pop_left:NNN #1#2#3
6678   { \exp_after:wN \__seq_pop_left:wnwNNN #2 \q_stop #1#2#3 }
6679 \cs_new_protected:Npn \__seq_pop_left:wnwNNN
6680     #1 \__seq_item:n #2#3 \q_stop #4#5#6
6681   {
6682     #4 #5 { #1 #3 }
6683     \tl_set:Nn #6 {#2}
6684   }
6685 \cs_generate_variant:Nn \seq_pop_left:NN  { c }
6686 \cs_generate_variant:Nn \seq_gpop_left:NN { c }
```

(*End definition for* \seq_pop_left:NN *and others. These functions are documented on page 114.*)

\seq_get_right:NN
\seq_get_right:cN
__seq_get_right_loop:nn

First remove \s__seq and prepend \q_no_value, then take two arguments at a time. Before the right-hand end of the sequence, this is a brace group followed by __seq_-item:n, both removed by \use_none:nn. At the end of the sequence, the two question marks are taken by \use_none:nn, and the assignment is placed before the right-most item. In the next iteration, __seq_get_right_loop:nn receives two empty arguments, and \use_none:nn stops the loop.

```
6687 \cs_new_protected:Npn \seq_get_right:NN #1#2
6688   {
6689     \exp_after:wN \use_i_ii:nnn
6690     \exp_after:wN \__seq_get_right_loop:nn
6691     \exp_after:wN \q_no_value
6692     #1
6693     { ?? \tl_set:Nn #2 }
6694     { } { }
6695   }
6696 \cs_new_protected:Npn \__seq_get_right_loop:nn #1#2
6697   {
6698     \use_none:nn #2 {#1}
6699     \__seq_get_right_loop:nn
6700   }
6701 \cs_generate_variant:Nn \seq_get_right:NN { c }
```

(*End definition for* \seq_get_right:NN *and* __seq_get_right_loop:nn. *These functions are documented on page 114.*)

\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 x-type expansion and a "non-expanding" definition for __seq_item:n, the left-most $n - 1$ entries in a sequence of n items will be 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

416

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.

```
6702 \cs_new_protected:Npn \seq_pop_right:NN
6703   { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_set:Nx }
6704 \cs_new_protected:Npn \seq_gpop_right:NN
6705   { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_gset:Nx }
6706 \cs_new_protected:Npn \__seq_pop_right:NNN #1#2#3
6707   {
6708     \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
6709     \cs_set_eq:NN \__seq_item:n \scan_stop:
6710     #1 #2
6711       { \if_false: } \fi: \s__seq
6712         \exp_after:wN \use_i:nnn
6713         \exp_after:wN \__seq_pop_right_loop:nn
6714         #2
6715         {
6716           \if_false: { \fi: }
6717           \tl_set:Nx #3
6718         }
6719         { } \use_none:nn
6720     \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
6721   }
6722 \cs_new:Npn \__seq_pop_right_loop:nn #1#2
6723   {
6724     #2 { \exp_not:n {#1} }
6725     \__seq_pop_right_loop:nn
6726   }
6727 \cs_generate_variant:Nn \seq_pop_right:NN  { c }
6728 \cs_generate_variant:Nn \seq_gpop_right:NN { c }
```

(*End definition for* `\seq_pop_right:NN` *and others. These functions are documented on page 114.*)

`\seq_get_left:NNTF`
`\seq_get_left:cNTF`
`\seq_get_right:NNTF`
`\seq_get_right:cNTF`

Getting from the left or right with a check on the results. The first argument to `\__-seq_pop_TF:NNNN` is left unused.

```
6729 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
6730   { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
6731 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
6732   { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
6733 \cs_generate_variant:Nn \seq_get_left:NNT   { c }
6734 \cs_generate_variant:Nn \seq_get_left:NNF   { c }
6735 \cs_generate_variant:Nn \seq_get_left:NNTF  { c }
6736 \cs_generate_variant:Nn \seq_get_right:NNT  { c }
6737 \cs_generate_variant:Nn \seq_get_right:NNF  { c }
6738 \cs_generate_variant:Nn \seq_get_right:NNTF { c }
```

(*End definition for* `\seq_get_left:NNTF` *and* `\seq_get_right:NNTF`. *These functions are documented on page 115.*)

`\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.

```
6739 \prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2 { T , F , TF }
6740   { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_set:Nn #1 #2 }
6741 \prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2 { T , F , TF }
6742   { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_gset:Nn #1 #2 }
6743 \prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2 { T , F , TF }
```

```
6744      { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \tl_set:Nx #1 #2 }
6745  \prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2 { T , F , TF }
6746      { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \tl_gset:Nx #1 #2 }
6747  \cs_generate_variant:Nn \seq_pop_left:NNT    { c }
6748  \cs_generate_variant:Nn \seq_pop_left:NNF    { c }
6749  \cs_generate_variant:Nn \seq_pop_left:NNTF   { c }
6750  \cs_generate_variant:Nn \seq_gpop_left:NNT   { c }
6751  \cs_generate_variant:Nn \seq_gpop_left:NNF   { c }
6752  \cs_generate_variant:Nn \seq_gpop_left:NNTF  { c }
6753  \cs_generate_variant:Nn \seq_pop_right:NNT   { c }
6754  \cs_generate_variant:Nn \seq_pop_right:NNF   { c }
6755  \cs_generate_variant:Nn \seq_pop_right:NNTF  { c }
6756  \cs_generate_variant:Nn \seq_gpop_right:NNT  { c }
6757  \cs_generate_variant:Nn \seq_gpop_right:NNF  { c }
6758  \cs_generate_variant:Nn \seq_gpop_right:NNTF { c }
```

(*End definition for* `\seq_pop_left:NNTF` *and others. These functions are documented on page 115.*)

`\seq_item:Nn`
`\seq_item:cn`
`\__seq_item:wNn`
`\__seq_item:nnn`

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 stop code `{ ? \__prg_break: } { }` will be used by the auxiliary, terminating the loop and returning nothing at all.

```
6759  \cs_new:Npn \seq_item:Nn #1
6760      { \exp_after:wN \__seq_item:wNn #1 \q_stop #1 }
6761  \cs_new:Npn \__seq_item:wNn \s__seq #1 \q_stop #2#3
6762      {
6763        \exp_args:Nf \__seq_item:nnn
6764          {
6765            \int_eval:n
6766              {
6767                \int_compare:nNnT {#3} < \c_zero
6768                  { \seq_count:N #2 + \c_one + }
6769                #3
6770              }
6771          }
6772        #1
6773        { ? \__prg_break: } { }
6774        \__prg_break_point:
6775      }
6776  \cs_new:Npn \__seq_item:nnn #1#2#3
6777      {
6778        \use_none:n #2
6779        \int_compare:nNnTF {#1} = \c_one
6780          { \__prg_break:n { \exp_not:n {#3} } }
6781          { \exp_args:Nf \__seq_item:nnn { \int_eval:n { #1 - 1 } } }
6782      }
6783  \cs_generate_variant:Nn \seq_item:Nn { c }
```

(*End definition for* `\seq_item:Nn` *,* `\__seq_item:wNn` *, and* `\__seq_item:nnn` *. These functions are documented on page 114.*)

12.6   Mapping to 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`

418

is inserted.

```
6784 \cs_new:Npn \seq_map_break:
6785     { \__prg_map_break:Nn \seq_map_break: { } }
6786 \cs_new:Npn \seq_map_break:n
6787     { \__prg_map_break:Nn \seq_map_break: }
```

(*End definition for* `\seq_map_break:` *and* `\seq_map_break:n`*. These functions are documented on page 117.*)

`\seq_map_function:NN`
`\seq_map_function:cN`
`\__seq_map_function:NNn`

The idea here is to apply the code of `#2` to each item in the sequence without altering the definition of `\__seq_item:n`. This is done as by noting that every odd token in the sequence must be `\__seq_item:n`, which can be gobbled by `\use_none:n`. At the end of the loop, `#2` is instead `? \seq_map_break:`, which therefore breaks the loop without needing to do a (relatively-expensive) quark test.

```
6788 \cs_new:Npn \seq_map_function:NN #1#2
6789     {
6790         \exp_after:wN \use_i_ii:nnn
6791         \exp_after:wN \__seq_map_function:NNn
6792         \exp_after:wN #2
6793         #1
6794         { ? \seq_map_break: } { }
6795         \__prg_break_point:Nn \seq_map_break: { }
6796     }
6797 \cs_new:Npn \__seq_map_function:NNn #1#2#3
6798     {
6799         \use_none:n #2
6800         #1 {#3}
6801         \__seq_map_function:NNn #1
6802     }
6803 \cs_generate_variant:Nn \seq_map_function:NN { c }
```

(*End definition for* `\seq_map_function:NN` *and* `\__seq_map_function:NNn`*. These functions are documented on page 116.*)

`\__seq_push_item_def:n`
`\__seq_push_item_def:x`
`\__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.

```
6804 \cs_new_protected:Npn \__seq_push_item_def:n
6805     {
6806         \__seq_push_item_def:
6807         \cs_gset:Npn \__seq_item:n ##1
6808     }
6809 \cs_new_protected:Npn \__seq_push_item_def:x
6810     {
6811         \__seq_push_item_def:
6812         \cs_gset:Npx \__seq_item:n ##1
6813     }
6814 \cs_new_protected:Npn \__seq_push_item_def:
6815     {
6816         \int_gincr:N \g__prg_map_int
6817         \cs_gset_eq:cN { __prg_map_ \int_use:N \g__prg_map_int :w }
6818             \__seq_item:n
6819     }
6820 \cs_new_protected:Npn \__seq_pop_item_def:
```

419

```
6821    {
6822      \cs_gset_eq:Nc \__seq_item:n
6823        { __prg_map_ \int_use:N \g__prg_map_int :w }
6824      \int_gdecr:N \g__prg_map_int
6825    }
```

(*End definition for* `\__seq_push_item_def:n`, `\__seq_push_item_def:`, *and* `\__seq_pop_item_def:`.)

`\seq_map_inline:Nn` The idea here is that `\__seq_item:n` is already "applied" to each item in a sequence,
`\seq_map_inline:cn` and so an in-line mapping is just a case of redefining `\__seq_item:n`.

```
6826 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
6827   {
6828     \__seq_push_item_def:n {#2}
6829     #1
6830     \__prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
6831   }
6832 \cs_generate_variant:Nn \seq_map_inline:Nn { c }
```

(*End definition for* `\seq_map_inline:Nn`. *This function is documented on page 116.*)

`\seq_map_variable:NNn` This is just a specialised version of the in-line mapping function, using an x-type expan-
`\seq_map_variable:Ncn` sion for the code set up so that the number of # tokens required is as expected.
`\seq_map_variable:cNn`
`\seq_map_variable:ccn`

```
6833 \cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
6834   {
6835     \__seq_push_item_def:x
6836       {
6837         \tl_set:Nn \exp_not:N #2 {##1}
6838         \exp_not:n {#3}
6839       }
6840     #1
6841     \__prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
6842   }
6843 \cs_generate_variant:Nn \seq_map_variable:NNn {     Nc }
6844 \cs_generate_variant:Nn \seq_map_variable:NNn { c , cc }
```

(*End definition for* `\seq_map_variable:NNn`. *This function is documented on page 117.*)

`\seq_count:N` Counting the items in a sequence is done using the same approach as for other count
`\seq_count:c` functions: turn each entry into a +1 then use integer evaluation to actually do the math-
`\__seq_count:n` ematics.

```
6845 \cs_new:Npn \seq_count:N #1
6846   {
6847     \int_eval:n
6848       {
6849         0
6850         \seq_map_function:NN #1 \__seq_count:n
6851       }
6852   }
6853 \cs_new:Npn \__seq_count:n #1 { + \c_one }
6854 \cs_generate_variant:Nn \seq_count:N { c }
```

(*End definition for* `\seq_count:N` *and* `\__seq_count:n`. *These functions are documented on page 118.*)

420

12.7 Using sequences

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`.

```
6855 \cs_new:Npn \seq_use:Nnnn #1#2#3#4
6856   {
6857     \seq_if_exist:NTF #1
6858       {
6859         \int_case:nnF { \seq_count:N #1 }
6860           {
6861             { 0 } { }
6862             { 1 } { \exp_after:wN \__seq_use:NNnNnn #1 ? { } { } }
6863             { 2 } { \exp_after:wN \__seq_use:NNnNnn #1 {#2} }
6864           }
6865           {
6866             \exp_after:wN \__seq_use_setup:w #1 \__seq_item:n
6867             \q_mark { \__seq_use:nwwwwnwn {#3} }
6868             \q_mark { \__seq_use:nwwn {#4} }
6869             \q_stop { }
6870           }
6871       }
6872       {
6873         \__msg_kernel_expandable_error:nnn
6874           { kernel } { bad-variable } {#1}
6875       }
6876   }
6877 \cs_generate_variant:Nn \seq_use:Nnnn { c }
6878 \cs_new:Npn \__seq_use:NNnNnn #1#2#3#4#5#6 { \exp_not:n { #3 #6 #5 } }
6879 \cs_new:Npn \__seq_use_setup:w \s__seq { \__seq_use:nwwwwnwn { } }
6880 \cs_new:Npn \__seq_use:nwwwwnwn
6881     #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4#5
6882     \q_mark #6#7 \q_stop #8
6883   {
6884     #6 \__seq_item:n {#3} \__seq_item:n {#4} #5
6885     \q_mark {#6} #7 \q_stop { #8 #1 #2 }
6886   }
6887 \cs_new:Npn \__seq_use:nwwn #1 \__seq_item:n #2 #3 \q_stop #4
6888   { \exp_not:n { #4 #1 #2 } }
6889 \cs_new:Npn \seq_use:Nn #1#2
6890   { \seq_use:Nnnn #1 {#2} {#2} {#2} }
6891 \cs_generate_variant:Nn \seq_use:Nn { c }
```

(*End definition for* `\seq_use:Nnnn` *and others. These functions are documented on page 118.*)

12.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

Pushing to a sequence is the same as adding on the left.

```
6892 \cs_new_eq:NN \seq_push:Nn  \seq_put_left:Nn
6893 \cs_new_eq:NN \seq_push:NV  \seq_put_left:NV
6894 \cs_new_eq:NN \seq_push:Nv  \seq_put_left:Nv
6895 \cs_new_eq:NN \seq_push:No  \seq_put_left:No
```

```
6896 \cs_new_eq:NN \seq_push:Nx  \seq_put_left:Nx
6897 \cs_new_eq:NN \seq_push:cn  \seq_put_left:cn
6898 \cs_new_eq:NN \seq_push:cV  \seq_put_left:cV
6899 \cs_new_eq:NN \seq_push:cv  \seq_put_left:cv
6900 \cs_new_eq:NN \seq_push:co  \seq_put_left:co
6901 \cs_new_eq:NN \seq_push:cx  \seq_put_left:cx
6902 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
6903 \cs_new_eq:NN \seq_gpush:NV \seq_gput_left:NV
6904 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
6905 \cs_new_eq:NN \seq_gpush:No \seq_gput_left:No
6906 \cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
6907 \cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
6908 \cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
6909 \cs_new_eq:NN \seq_gpush:cv \seq_gput_left:cv
6910 \cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
6911 \cs_new_eq:NN \seq_gpush:cx \seq_gput_left:cx
```

(*End definition for* `\seq_push:Nn` *and* `\seq_gpush:Nn`. *These functions are documented on page 119.*)

\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.

```
6912 \cs_new_eq:NN \seq_get:NN \seq_get_left:NN
6913 \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
6914 \cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
6915 \cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
6916 \cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
6917 \cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN
```

(*End definition for* `\seq_get:NN`, `\seq_pop:NN`, *and* `\seq_gpop:NN`. *These functions are documented on page 119.*)

\seq_get:NN*TF*
\seq_get:cN*TF*
\seq_pop:NN*TF*
\seq_pop:cN*TF*
\seq_gpop:NN*TF*
\seq_gpop:cN*TF*

More copies.

```
6918 \prg_new_eq_conditional:NNn \seq_get:NN  \seq_get_left:NN  { T , F , TF }
6919 \prg_new_eq_conditional:NNn \seq_get:cN  \seq_get_left:cN  { T , F , TF }
6920 \prg_new_eq_conditional:NNn \seq_pop:NN  \seq_pop_left:NN  { T , F , TF }
6921 \prg_new_eq_conditional:NNn \seq_pop:cN  \seq_pop_left:cN  { T , F , TF }
6922 \prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
6923 \prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }
```

(*End definition for* `\seq_get:NNTF`, `\seq_pop:NNTF`, *and* `\seq_gpop:NNTF`. *These functions are documented on page 119.*)

12.9   Viewing sequences

\seq_show:N
\seq_show:c

Apply the general `\__msg_show_variable:NNNnn`.

```
6924 \cs_new_protected:Npn \seq_show:N #1
6925   {
6926     \__msg_show_variable:NNNnn #1
6927       \seq_if_exist:NTF \seq_if_empty:NTF { seq }
6928       { \seq_map_function:NN #1 \__msg_show_item:n }
6929   }
6930 \cs_generate_variant:Nn \seq_show:N { c }
```

(*End definition for* `\seq_show:N`. *This function is documented on page 121.*)

12.10 Scratch sequences

\l_tmpa_seq Temporary comma list variables.
\l_tmpb_seq
\g_tmpa_seq
\g_tmpb_seq

```
6931 \seq_new:N \l_tmpa_seq
6932 \seq_new:N \l_tmpb_seq
6933 \seq_new:N \g_tmpa_seq
6934 \seq_new:N \g_tmpb_seq
```

(*End definition for* \l_tmpa_seq *and others. These variables are documented on page 121.*)

```
6935 ⟨/initex | package⟩
```

13 l3clist implementation

The following test files are used for this code: m3clist002.

```
6936 ⟨*initex | package⟩
```

```
6937 ⟨@@=clist⟩
```

\c_empty_clist An empty comma list is simply an empty token list.

```
6938 \cs_new_eq:NN \c_empty_clist \c_empty_tl
```

(*End definition for* \c_empty_clist. *This variable is documented on page 130.*)

\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

```
6939 \tl_new:N \l__clist_internal_clist
```

(*End definition for* \l__clist_internal_clist.)

__clist_tmp:w A temporary function for various purposes.

```
6940 \cs_new_protected:Npn \__clist_tmp:w { }
```

(*End definition for* __clist_tmp:w.)

13.1 Allocation and initialisation

\clist_new:N Internally, comma lists are just token lists.
\clist_new:c

```
6941 \cs_new_eq:NN \clist_new:N \tl_new:N
6942 \cs_new_eq:NN \clist_new:c \tl_new:c
```

(*End definition for* \clist_new:N. *This function is documented on page 123.*)

\clist_const:Nn Creating and initializing a constant comma list is done in a way similar to \clist_set:Nn
\clist_const:cn and \clist_gset:Nn, being careful to strip spaces.
\clist_const:Nx
\clist_const:cx

```
6943 \cs_new_protected:Npn \clist_const:Nn #1#2
6944   { \tl_const:Nx #1 { \__clist_trim_spaces:n {#2} } }
6945 \cs_generate_variant:Nn \clist_const:Nn { c , Nx , cx }
```

(*End definition for* \clist_const:Nn. *This function is documented on page 123.*)

\clist_clear:N Clearing comma lists is just the same as clearing token lists.
\clist_clear:c
\clist_gclear:N
\clist_gclear:c

```
6946 \cs_new_eq:NN \clist_clear:N  \tl_clear:N
6947 \cs_new_eq:NN \clist_clear:c  \tl_clear:c
6948 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
6949 \cs_new_eq:NN \clist_gclear:c \tl_gclear:c
```

(*End definition for* `\clist_clear:N` *and* `\clist_gclear:N`. *These functions are documented on page* *123*.)

`\clist_clear_new:N` Once again a copy from the token list functions.
`\clist_clear_new:c`
`\clist_gclear_new:N`
`\clist_gclear_new:c`

```
6950 \cs_new_eq:NN \clist_clear_new:N  \tl_clear_new:N
6951 \cs_new_eq:NN \clist_clear_new:c  \tl_clear_new:c
6952 \cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
6953 \cs_new_eq:NN \clist_gclear_new:c \tl_gclear_new:c
```

(*End definition for* `\clist_clear_new:N` *and* `\clist_gclear_new:N`. *These functions are documented on page* *123*.)

`\clist_set_eq:NN` Once again, these are simple copies from the token list functions.
`\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`

```
6954 \cs_new_eq:NN \clist_set_eq:NN  \tl_set_eq:NN
6955 \cs_new_eq:NN \clist_set_eq:Nc  \tl_set_eq:Nc
6956 \cs_new_eq:NN \clist_set_eq:cN  \tl_set_eq:cN
6957 \cs_new_eq:NN \clist_set_eq:cc  \tl_set_eq:cc
6958 \cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
6959 \cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
6960 \cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
6961 \cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc
```

(*End definition for* `\clist_set_eq:NN` *and* `\clist_gset_eq:NN`. *These functions are documented on page* *124*.)

`\clist_set_from_seq:NN` Setting a comma list from a comma-separated list is done using a simple mapping. We
`\clist_set_from_seq:cN` wrap most items with `\exp_not:n`, and a comma. Items which contain a comma or a
`\clist_set_from_seq:Nc` space are surrounded by an extra set of braces. The first comma must be removed, except
`\clist_set_from_seq:cc` in the case of an empty comma-list.
`\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_wrap_item:n`
`\__clist_set_from_seq:w`

```
6962 \cs_new_protected:Npn \clist_set_from_seq:NN
6963   { \__clist_set_from_seq:NNNN \clist_clear:N  \tl_set:Nx  }
6964 \cs_new_protected:Npn \clist_gset_from_seq:NN
6965   { \__clist_set_from_seq:NNNN \clist_gclear:N \tl_gset:Nx }
6966 \cs_new_protected:Npn \__clist_set_from_seq:NNNN #1#2#3#4
6967   {
6968     \seq_if_empty:NTF #4
6969       { #1 #3 }
6970       {
6971         #2 #3
6972           {
6973             \exp_last_unbraced:Nf \use_none:n
6974               { \seq_map_function:NN #4 \__clist_wrap_item:n }
6975           }
6976       }
6977   }
6978 \cs_new:Npn \__clist_wrap_item:n #1
6979   {
6980     ,
6981     \tl_if_empty:oTF { \__clist_set_from_seq:w #1 ~ , #1 ~ }
6982       { \exp_not:n   {#1}    }
6983       { \exp_not:n { {#1} } }
6984   }
6985 \cs_new:Npn \__clist_set_from_seq:w #1 , #2 ~ { }
6986 \cs_generate_variant:Nn \clist_set_from_seq:NN  {     Nc }
```

424

```
6987 \cs_generate_variant:Nn \clist_set_from_seq:NN  { c , cc }
6988 \cs_generate_variant:Nn \clist_gset_from_seq:NN {     Nc }
6989 \cs_generate_variant:Nn \clist_gset_from_seq:NN { c , cc }
```

(*End definition for* \clist_set_from_seq:NN *and others. These functions are documented on page [124].*)

\clist_concat:NNN Concatenating comma lists is not quite as easy as it seems, as there needs to be the
\clist_concat:ccc correct addition of a comma to the output. So a little work to do.
\clist_gconcat:NNN
\clist_gconcat:ccc
__clist_concat:NNNN

```
6990 \cs_new_protected:Npn \clist_concat:NNN
6991   { \__clist_concat:NNNN \tl_set:Nx }
6992 \cs_new_protected:Npn \clist_gconcat:NNN
6993   { \__clist_concat:NNNN \tl_gset:Nx }
6994 \cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4
6995   {
6996     #1 #2
6997       {
6998         \exp_not:o #3
6999         \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
7000         \exp_not:o #4
7001       }
7002   }
7003 \cs_generate_variant:Nn \clist_concat:NNN  { ccc }
7004 \cs_generate_variant:Nn \clist_gconcat:NNN { ccc }
```

(*End definition for* \clist_concat:NNN, \clist_gconcat:NNN, *and* __clist_concat:NNNN. *These functions are documented on page [124].*)

\clist_if_exist_p:N Copies of the cs functions defined in l3basics.
\clist_if_exist_p:c
\clist_if_exist:N*TF*
\clist_if_exist:c*TF*

```
7005 \prg_new_eq_conditional:NNn \clist_if_exist:N \cs_if_exist:N
7006   { TF , T , F , p }
7007 \prg_new_eq_conditional:NNn \clist_if_exist:c \cs_if_exist:c
7008   { TF , T , F , p }
```

(*End definition for* \clist_if_exist:NTF. *This function is documented on page [124].*)

13.2   Removing spaces around items

__clist_trim_spaces_generic:nw This expands to the ⟨*code*⟩, followed by a brace group containing the ⟨*item*⟩, with leading
__clist_trim_spaces_generic:nn and trailing spaces removed. The calling function is responsible for inserting \q_mark
in front of the ⟨*item*⟩, as well as testing for the end of the list. We reuse a l3tl internal
function, whose first argument must start with \q_mark. That trims the item #2, then
feeds the result (after having to do an o-type expansion) to __clist_trim_spaces_-
generic:nn which places the ⟨*code*⟩ in front of the ⟨*trimmed item*⟩.

```
7009 \cs_new:Npn \__clist_trim_spaces_generic:nw #1#2 ,
7010   {
7011     \__tl_trim_spaces:nn {#2}
7012       { \exp_args:No \__clist_trim_spaces_generic:nn } {#1}
7013   }
7014 \cs_new:Npn \__clist_trim_spaces_generic:nn #1#2 { #2 {#1} }
```

(*End definition for* __clist_trim_spaces_generic:nw *and* __clist_trim_spaces_generic:nn.)

425

The first argument of __clist_trim_spaces:nn is initially empty, and later a comma, namely, as soon as we have added an item to the resulting list. The auxiliary tests for the end of the list, and also prevents empty arguments from finding their way into the output.

```
7015 \cs_new:Npn \__clist_trim_spaces:n #1
7016   {
7017     \__clist_trim_spaces_generic:nw
7018       { \__clist_trim_spaces:nn { } }
7019     \q_mark #1 ,
7020     \q_recursion_tail, \q_recursion_stop
7021   }
7022 \cs_new:Npn \__clist_trim_spaces:nn #1 #2
7023   {
7024     \quark_if_recursion_tail_stop:n {#2}
7025     \tl_if_empty:nTF {#2}
7026       {
7027         \__clist_trim_spaces_generic:nw
7028           { \__clist_trim_spaces:nn {#1} } \q_mark
7029       }
7030       {
7031         #1 \exp_not:n {#2}
7032         \__clist_trim_spaces_generic:nw
7033           { \__clist_trim_spaces:nn { , } } \q_mark
7034       }
7035   }
```

(*End definition for* __clist_trim_spaces:n *and* __clist_trim_spaces:nn.)

13.3   Adding data to comma lists

\clist_set:Nn
\clist_set:NV
\clist_set:No
\clist_set:Nx
\clist_set:cn
\clist_set:cV
\clist_set:co
\clist_set:cx
\clist_gset:Nn
\clist_gset:NV
\clist_gset:No
\clist_gset:Nx
\clist_gset:cn
\clist_gset:cV
\clist_gset:co
\clist_gset:cx

```
7036 \cs_new_protected:Npn \clist_set:Nn #1#2
7037   { \tl_set:Nx #1 { \__clist_trim_spaces:n {#2} } }
7038 \cs_new_protected:Npn \clist_gset:Nn #1#2
7039   { \tl_gset:Nx #1 { \__clist_trim_spaces:n {#2} } }
7040 \cs_generate_variant:Nn \clist_set:Nn  { NV , No , Nx , c , cV , co , cx }
7041 \cs_generate_variant:Nn \clist_gset:Nn { NV , No , Nx , c , cV , co , cx }
```

(*End definition for* \clist_set:Nn *and* \clist_gset:Nn. *These functions are documented on page 124.*)

\clist_put_left:Nn
\clist_put_left:NV
\clist_put_left:No
\clist_put_left:Nx
\clist_put_left:cn
\clist_put_left:cV
\clist_put_left:co
\clist_put_left:cx
\clist_gput_left:Nn
\clist_gput_left:NV
\clist_gput_left:No
\clist_gput_left:Nx
\clist_gput_left:cn
\clist_gput_left:cV
\clist_gput_left:co
\clist_gput_left:cx
__clist_put_left:NNNn

Comma lists cannot hold empty values: there are therefore a couple of sanity checks to avoid accumulating commas.

```
7042 \cs_new_protected:Npn \clist_put_left:Nn
7043   { \__clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }
7044 \cs_new_protected:Npn \clist_gput_left:Nn
7045   { \__clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
7046 \cs_new_protected:Npn \__clist_put_left:NNNn #1#2#3#4
7047   {
7048     #2 \l__clist_internal_clist {#4}
7049     #1 #3 \l__clist_internal_clist #3
7050   }
7051 \cs_generate_variant:Nn \clist_put_left:Nn  {     NV , No , Nx }
7052 \cs_generate_variant:Nn \clist_put_left:Nn  { c , cV , co , cx }
7053 \cs_generate_variant:Nn \clist_gput_left:Nn {     NV , No , Nx }
```

```
7054 \cs_generate_variant:Nn \clist_gput_left:Nn { c , cV , co , cx }
```

(*End definition for* `\clist_put_left:Nn`, `\clist_gput_left:Nn`, *and* `\__clist_put_left:NNNn`. *These functions are documented on page 124.*)

```
7055 \cs_new_protected:Npn \clist_put_right:Nn
7056   { \__clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
7057 \cs_new_protected:Npn \clist_gput_right:Nn
7058   { \__clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
7059 \cs_new_protected:Npn \__clist_put_right:NNNn #1#2#3#4
7060   {
7061     #2 \l__clist_internal_clist {#4}
7062     #1 #3 #3 \l__clist_internal_clist
7063   }
7064 \cs_generate_variant:Nn \clist_put_right:Nn {     NV , No , Nx }
7065 \cs_generate_variant:Nn \clist_put_right:Nn  { c , cV , co , cx }
7066 \cs_generate_variant:Nn \clist_gput_right:Nn {     NV , No , Nx }
7067 \cs_generate_variant:Nn \clist_gput_right:Nn { c , cV , co , cx }
```

(*End definition for* `\clist_put_right:Nn`, `\clist_gput_right:Nn`, *and* `\__clist_put_right:NNNn`. *These functions are documented on page 125.*)

13.4 Comma lists as stacks

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma.

```
7068 \cs_new_protected:Npn \clist_get:NN #1#2
7069   {
7070     \if_meaning:w #1 \c_empty_clist
7071       \tl_set:Nn #2 { \q_no_value }
7072     \else:
7073       \exp_after:wN \__clist_get:wN #1 , \q_stop #2
7074     \fi:
7075   }
7076 \cs_new_protected:Npn \__clist_get:wN #1 , #2 \q_stop #3
7077   { \tl_set:Nn #3 {#1} }
7078 \cs_generate_variant:Nn \clist_get:NN { c }
```

(*End definition for* `\clist_get:NN` *and* `\__clist_get:wN`. *These functions are documented on page 129.*)

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 `\q_mark`, unless the original clist contained exactly one item: then the argument is just `\q_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.

```
7079 \cs_new_protected:Npn \clist_pop:NN
7080   { \__clist_pop:NNN \tl_set:Nx }
7081 \cs_new_protected:Npn \clist_gpop:NN
7082   { \__clist_pop:NNN \tl_gset:Nx }
7083 \cs_new_protected:Npn \__clist_pop:NNN #1#2#3
7084   {
7085     \if_meaning:w #2 \c_empty_clist
7086       \tl_set:Nn #3 { \q_no_value }
```

```
7087       \else:
7088         \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
7089       \fi:
7090   }
7091 \cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \q_stop #3#4#5
7092   {
7093     \tl_set:Nn #5 {#1}
7094     #3 #4
7095       {
7096         \__clist_pop:wN \prg_do_nothing:
7097           #2 \exp_not:o
7098             , \q_mark \use_none:n
7099           \q_stop
7100       }
7101   }
7102 \cs_new:Npn \__clist_pop:wN #1 , \q_mark #2 #3 \q_stop { #2 {#1} }
7103 \cs_generate_variant:Nn \clist_pop:NN  { c }
7104 \cs_generate_variant:Nn \clist_gpop:NN { c }
```

(*End definition for* `\clist_pop:NN` *and others. These functions are documented on page 129.*)

`\clist_get:NNTF`
`\clist_get:cNTF`
`\clist_pop:NNTF`
`\clist_pop:cNTF`
`\clist_gpop:NNTF`
`\clist_gpop:cNTF`
`\__clist_pop_TF:NNN`

The same, as branching code: very similar to the above.

```
7105 \prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
7106   {
7107     \if_meaning:w #1 \c_empty_clist
7108       \prg_return_false:
7109     \else:
7110       \exp_after:wN \__clist_get:wN #1 , \q_stop #2
7111       \prg_return_true:
7112     \fi:
7113   }
7114 \cs_generate_variant:Nn \clist_get:NNT  { c }
7115 \cs_generate_variant:Nn \clist_get:NNF  { c }
7116 \cs_generate_variant:Nn \clist_get:NNTF { c }
7117 \prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
7118   { \__clist_pop_TF:NNN \tl_set:Nx #1 #2 }
7119 \prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
7120   { \__clist_pop_TF:NNN \tl_gset:Nx #1 #2 }
7121 \cs_new_protected:Npn \__clist_pop_TF:NNN #1#2#3
7122   {
7123     \if_meaning:w #2 \c_empty_clist
7124       \prg_return_false:
7125     \else:
7126       \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
7127       \prg_return_true:
7128     \fi:
7129   }
7130 \cs_generate_variant:Nn \clist_pop:NNT   { c }
7131 \cs_generate_variant:Nn \clist_pop:NNF   { c }
7132 \cs_generate_variant:Nn \clist_pop:NNTF  { c }
7133 \cs_generate_variant:Nn \clist_gpop:NNT  { c }
7134 \cs_generate_variant:Nn \clist_gpop:NNF  { c }
7135 \cs_generate_variant:Nn \clist_gpop:NNTF { c }
```

(*End definition for* `\clist_get:NNTF` *and others. These functions are documented on page 129.*)

\clist_push:Nn	Pushing to a comma list is the same as adding on the left.

```
7136 \cs_new_eq:NN \clist_push:Nn  \clist_put_left:Nn
7137 \cs_new_eq:NN \clist_push:NV  \clist_put_left:NV
7138 \cs_new_eq:NN \clist_push:No  \clist_put_left:No
7139 \cs_new_eq:NN \clist_push:Nx  \clist_put_left:Nx
7140 \cs_new_eq:NN \clist_push:cn  \clist_put_left:cn
7141 \cs_new_eq:NN \clist_push:cV  \clist_put_left:cV
7142 \cs_new_eq:NN \clist_push:co  \clist_put_left:co
7143 \cs_new_eq:NN \clist_push:cx  \clist_put_left:cx
7144 \cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
7145 \cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV
7146 \cs_new_eq:NN \clist_gpush:No \clist_gput_left:No
7147 \cs_new_eq:NN \clist_gpush:Nx \clist_gput_left:Nx
7148 \cs_new_eq:NN \clist_gpush:cn \clist_gput_left:cn
7149 \cs_new_eq:NN \clist_gpush:cV \clist_gput_left:cV
7150 \cs_new_eq:NN \clist_gpush:co \clist_gput_left:co
7151 \cs_new_eq:NN \clist_gpush:cx \clist_gput_left:cx
```

The functions \clist_push:Nn are:
\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

(*End definition for* \clist_push:Nn *and* \clist_gpush:Nn*. These functions are documented on page 130.*)

13.5 Modifying comma lists

\l__clist_internal_remove_clist An internal comma list for the removal routines.

```
7152 \clist_new:N \l__clist_internal_remove_clist
```

(*End definition for* \l__clist_internal_remove_clist*.*)

\clist_remove_duplicates:N Removing duplicates means making a new list then copying it.
\clist_remove_duplicates:c
\clist_gremove_duplicates:N
\clist_gremove_duplicates:c
__clist_remove_duplicates:NN

```
7153 \cs_new_protected:Npn \clist_remove_duplicates:N
7154   { \__clist_remove_duplicates:NN \clist_set_eq:NN }
7155 \cs_new_protected:Npn \clist_gremove_duplicates:N
7156   { \__clist_remove_duplicates:NN \clist_gset_eq:NN }
7157 \cs_new_protected:Npn \__clist_remove_duplicates:NN #1#2
7158   {
7159     \clist_clear:N \l__clist_internal_remove_clist
7160     \clist_map_inline:Nn #2
7161       {
7162         \clist_if_in:NnF \l__clist_internal_remove_clist {##1}
7163           { \clist_put_right:Nn \l__clist_internal_remove_clist {##1} }
7164       }
7165     #1 #2 \l__clist_internal_remove_clist
7166   }
7167 \cs_generate_variant:Nn \clist_remove_duplicates:N { c }
7168 \cs_generate_variant:Nn \clist_gremove_duplicates:N { c }
```

(*End definition for* \clist_remove_duplicates:N*,* \clist_gremove_duplicates:N*, and* __clist_-remove_duplicates:NN*. These functions are documented on page 125.*)

\clist_remove_all:Nn The method used here is very similar to \tl_replace_all:Nnn. Build a function de-
\clist_remove_all:cn limited by the ⟨*item*⟩ that should be removed, surrounded with commas, and call that
\clist_gremove_all:Nn function followed by the expanded comma list, and another copy of the ⟨*item*⟩. The loop
\clist_gremove_all:cn is controlled by the argument grabbed by __clist_remove_all:w: when the item was
__clist_remove_all:NNn
__clist_remove_all:w
__clist_remove_all:

found, the `\q_mark` delimiter used is the one inserted by `\__clist_tmp:w`, and `\use_-`
`none_delimit_by_q_stop:w` is deleted. At the end, the final ⟨*item*⟩ is grabbed, and
the argument of `\__clist_tmp:w` contains `\q_mark`: in that case, `\__clist_remove_-`
`all:w` removes the second `\q_mark` (inserted by `\__clist_tmp:w`), and lets `\use_none_-`
`delimit_by_q_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.

```
7169 \cs_new_protected:Npn \clist_remove_all:Nn
7170   { \__clist_remove_all:NNn \tl_set:Nx }
7171 \cs_new_protected:Npn \clist_gremove_all:Nn
7172   { \__clist_remove_all:NNn \tl_gset:Nx }
7173 \cs_new_protected:Npn \__clist_remove_all:NNn #1#2#3
7174   {
7175     \cs_set:Npn \__clist_tmp:w ##1 , #3 ,
7176       {
7177         ##1
7178         , \q_mark , \use_none_delimit_by_q_stop:w ,
7179         \__clist_remove_all:
7180       }
7181     #1 #2
7182       {
7183         \exp_after:wN \__clist_remove_all:
7184         #2 , \q_mark , #3 , \q_stop
7185       }
7186     \clist_if_empty:NF #2
7187       {
7188         #1 #2
7189           {
7190             \exp_args:No \exp_not:o
7191               { \exp_after:wN \use_none:n #2 }
7192           }
7193       }
7194   }
7195 \cs_new:Npn \__clist_remove_all:
7196   { \exp_after:wN \__clist_remove_all:w \__clist_tmp:w , }
7197 \cs_new:Npn \__clist_remove_all:w #1 , \q_mark , #2 , { \exp_not:n {#1} }
7198 \cs_generate_variant:Nn \clist_remove_all:Nn  { c }
7199 \cs_generate_variant:Nn \clist_gremove_all:Nn { c }
```

(*End definition for* `\clist_remove_all:Nn` *and others. These functions are documented on page 125.*)

`\clist_reverse:N`
`\clist_reverse:c`
`\clist_greverse:N`
`\clist_greverse:c`

Use `\clist_reverse:n` in an x-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.

```
7200 \cs_new_protected:Npn \clist_reverse:N #1
7201   { \tl_set:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
7202 \cs_new_protected:Npn \clist_greverse:N #1
7203   { \tl_gset:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
7204 \cs_generate_variant:Nn \clist_reverse:N { c }
7205 \cs_generate_variant:Nn \clist_greverse:N { c }
```

(End definition for `\clist_reverse:N` *and* `\clist_greverse:N`*. These functions are documented on page [125](#).)*

`\clist_reverse:n`
`\__clist_reverse:wwNww`
`\__clist_reverse_end:ww`

The reversed token list is built one item at a time, and stored between `\q_stop` and `\q_mark`, in the form of `?` followed by zero or more instances of "⟨*item*⟩,". We start from a comma list "⟨*item*₁⟩,...,⟨*item*ₙ⟩". During the loop, the auxiliary `\__clist_-reverse:wwNww` receives "?⟨*item*ᵢ⟩" as `#1`, "⟨*item*ᵢ₊₁⟩,...,⟨*item*ₙ⟩" as `#2`, `\__clist_-reverse:wwNww` as `#3`, what remains until `\q_stop` as `#4`, and "⟨*item*ᵢ₋₁⟩,...,⟨*item*₁⟩," 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 "`\q_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 `\q_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`.

```
7206 \cs_new:Npn \clist_reverse:n #1
7207   {
7208     \__clist_reverse:wwNww ? #1 ,
7209       \q_mark \__clist_reverse:wwNww ! ,
7210       \q_mark \__clist_reverse_end:ww
7211       \q_stop ? \q_mark
7212   }
7213 \cs_new:Npn \__clist_reverse:wwNww
7214     #1 , #2 \q_mark #3 #4 \q_stop ? #5 \q_mark
7215   { #3 ? #2 \q_mark #3 #4 \q_stop #1 , #5 \q_mark }
7216 \cs_new:Npn \__clist_reverse_end:ww #1 ! #2 , \q_mark
7217   { \exp_not:o { \use_none:n #2 } }
```

(End definition for `\clist_reverse:n`*,* `\__clist_reverse:wwNww`*, and* `\__clist_reverse_end:ww`*. These functions are documented on page [125](#).)*

13.6 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.

```
7218 \prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N
7219   { p , T , F , TF }
7220 \prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c
7221   { p , T , F , TF }
```

(End definition for `\clist_if_empty:NTF`*. This function is documented on page [126](#).)*

`\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 will grab `\prg_return_false:` as `#2`, unless every item in the comma list was blank and the loop actually got broken by the trailing `\q_mark \prg_return_false:` item.

```
7222 \prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
7223   {
7224     \__clist_if_empty_n:w ? #1
7225     , \q_mark \prg_return_false:
7226     , \q_mark \prg_return_true:
```

431

```
7227        \q_stop
7228      }
7229  \cs_new:Npn \__clist_if_empty_n:w #1 ,
7230      {
7231        \tl_if_empty:oTF { \use_none:nn #1 ? }
7232          { \__clist_if_empty_n:w ? }
7233          { \__clist_if_empty_n:wNw }
7234      }
7235  \cs_new:Npn \__clist_if_empty_n:wNw #1 \q_mark #2#3 \q_stop {#2}
```

(*End definition for* \clist_if_empty:nTF, __clist_if_empty_n:w, *and* __clist_if_empty_n:wNw. *These functions are documented on page 126.*)

\clist_if_in:NnTF See description of the \tl_if_in:Nn function for details. We simply surround the comma
\clist_if_in:NVTF list, and the item, with commas.
\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:nn

```
7236  \prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T  , F , TF }
7237      {
7238        \exp_args:No \__clist_if_in_return:nn #1 {#2}
7239      }
7240  \prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T  , F , TF }
7241      {
7242        \clist_set:Nn \l__clist_internal_clist {#1}
7243        \exp_args:No \__clist_if_in_return:nn \l__clist_internal_clist {#2}
7244      }
7245  \cs_new_protected:Npn \__clist_if_in_return:nn #1#2
7246      {
7247        \cs_set:Npn \__clist_tmp:w ##1 ,#2, { }
7248        \tl_if_empty:oTF
7249          { \__clist_tmp:w ,#1, {} {} ,#2, }
7250          { \prg_return_false: } { \prg_return_true: }
7251      }
7252  \cs_generate_variant:Nn \clist_if_in:NnT  {     NV , No }
7253  \cs_generate_variant:Nn \clist_if_in:NnT  { c , cV , co }
7254  \cs_generate_variant:Nn \clist_if_in:NnF  {     NV , No }
7255  \cs_generate_variant:Nn \clist_if_in:NnF  { c , cV , co }
7256  \cs_generate_variant:Nn \clist_if_in:NnTF {     NV , No }
7257  \cs_generate_variant:Nn \clist_if_in:NnTF { c , cV , co }
7258  \cs_generate_variant:Nn \clist_if_in:nnT  {     nV , no }
7259  \cs_generate_variant:Nn \clist_if_in:nnF  {     nV , no }
7260  \cs_generate_variant:Nn \clist_if_in:nnTF {     nV , no }
```

(*End definition for* \clist_if_in:NnTF, \clist_if_in:nnTF, *and* __clist_if_in_return:nn. *These functions are documented on page 126.*)

13.7 Mapping to comma lists

\clist_map_function:NN If the variable is empty, the mapping is skipped (otherwise, that comma-list would be
\clist_map_function:cN seen as consisting of one empty item). Then loop over the comma-list, grabbing one
__clist_map_function:Nw comma-delimited item at a time. The end is marked by \q_recursion_tail. The aux-
iliary function __clist_map_function:Nw is used directly in \clist_map_inline:Nn.
Change with care.

```
7261  \cs_new:Npn \clist_map_function:NN #1#2
7262      {
7263        \clist_if_empty:NF #1
```

432

```
7264            {
7265              \exp_last_unbraced:NNo \__clist_map_function:Nw #2 #1
7266                , \q_recursion_tail ,
7267              \__prg_break_point:Nn \clist_map_break: { }
7268            }
7269        }
7270  \cs_new:Npn \__clist_map_function:Nw #1#2 ,
7271        {
7272          \__quark_if_recursion_tail_break:nN {#2} \clist_map_break:
7273          #1 {#2}
7274          \__clist_map_function:Nw #1
7275        }
7276  \cs_generate_variant:Nn \clist_map_function:NN { c }
```

(*End definition for* \clist_map_function:NN *and* __clist_map_function:Nw. *These functions are documented on page* *126.*)

\clist_map_function:nN
__clist_map_function_n:Nn
__clist_map_unbrace:Nw

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_spaces_generic:nw. The auxiliary __clist_map_function_n:Nn receives as arguments the function, and the result of removing leading and trailing spaces from the item which lies until the next comma. Empty items are ignored, then one level of braces is removed by __clist_-map_unbrace:Nw.

```
7277  \cs_new:Npn \clist_map_function:nN #1#2
7278      {
7279        \__clist_trim_spaces_generic:nw { \__clist_map_function_n:Nn #2 }
7280        \q_mark #1, \q_recursion_tail,
7281        \__prg_break_point:Nn \clist_map_break: { }
7282      }
7283  \cs_new:Npn \__clist_map_function_n:Nn #1 #2
7284      {
7285        \__quark_if_recursion_tail_break:nN {#2} \clist_map_break:
7286        \tl_if_empty:nF {#2} { \__clist_map_unbrace:Nw #1 #2, }
7287        \__clist_trim_spaces_generic:nw { \__clist_map_function_n:Nn #1 }
7288        \q_mark
7289      }
7290  \cs_new:Npn \__clist_map_unbrace:Nw #1 #2, { #1 {#2} }
```

(*End definition for* \clist_map_function:nN, __clist_map_function_n:Nn, *and* __clist_map_unbrace:Nw. *These functions are documented on page* *126.*)

\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.

```
7291  \cs_new_protected:Npn \clist_map_inline:Nn #1#2
7292      {
7293        \clist_if_empty:NF #1
7294          {
7295            \int_gincr:N \g__prg_map_int
7296            \cs_gset:cpn { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}
7297            \exp_last_unbraced:Nco \__clist_map_function:Nw
```

433

```
7298              { __prg_map_ \int_use:N \g__prg_map_int :w }
7299              #1 , \q_recursion_tail ,
7300            \__prg_break_point:Nn \clist_map_break:
7301              { \int_gdecr:N \g__prg_map_int }
7302          }
7303      }
7304  \cs_new_protected:Npn \clist_map_inline:nn #1
7305      {
7306        \clist_set:Nn \l__clist_internal_clist {#1}
7307        \clist_map_inline:Nn \l__clist_internal_clist
7308      }
7309  \cs_generate_variant:Nn \clist_map_inline:Nn { c }
```

(*End definition for* \clist_map_inline:Nn *and* \clist_map_inline:nn. *These functions are documented on page 127.*)

\clist_map_variable:NNn   As for other comma-list mappings, filter out the case of an empty list. Same approach
\clist_map_variable:cNn   as \clist_map_function:Nn, additionally we store each item in the given variable. As
\clist_map_variable:nNn   for inline mappings, space trimming for the n variant is done by storing the comma list
__clist_map_variable:Nnw   in a variable.

```
7310  \cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
7311      {
7312        \clist_if_empty:NF #1
7313          {
7314            \exp_args:Nno \use:nn
7315              { \__clist_map_variable:Nnw #2 {#3} }
7316              #1
7317              , \q_recursion_tail , \q_recursion_stop
7318            \__prg_break_point:Nn \clist_map_break: { }
7319          }
7320      }
7321  \cs_new_protected:Npn \clist_map_variable:nNn #1
7322      {
7323        \clist_set:Nn \l__clist_internal_clist {#1}
7324        \clist_map_variable:NNn \l__clist_internal_clist
7325      }
7326  \cs_new_protected:Npn \__clist_map_variable:Nnw #1#2#3,
7327      {
7328        \tl_set:Nn #1 {#3}
7329        \quark_if_recursion_tail_stop:N #1
7330        \use:n {#2}
7331        \__clist_map_variable:Nnw #1 {#2}
7332      }
7333  \cs_generate_variant:Nn \clist_map_variable:NNn { c }
```

(*End definition for* \clist_map_variable:NNn, \clist_map_variable:nNn, *and* __clist_map_variable:Nnw. *These functions are documented on page 127.*)

\clist_map_break: The break statements use the general __prg_map_break:Nn mechanism.
\clist_map_break:n
```
7334  \cs_new:Npn \clist_map_break:
7335      { \__prg_map_break:Nn \clist_map_break: { } }
7336  \cs_new:Npn \clist_map_break:n
7337      { \__prg_map_break:Nn \clist_map_break: }
```

(End definition for `\clist_map_break:` *and* `\clist_map_break:n`. *These functions are documented on page 127.)*

<div style="margin-left: 2em;">

`\clist_count:N`
`\clist_count:c`
`\clist_count:n`
`\__clist_count:n`
`\__clist_count:w`

</div>

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).

```
7338 \cs_new:Npn \clist_count:N #1
7339   {
7340     \int_eval:n
7341       {
7342         0
7343         \clist_map_function:NN #1 \__clist_count:n
7344       }
7345   }
7346 \cs_generate_variant:Nn \clist_count:N { c }
7347 \cs_new:Npx \clist_count:n #1
7348   {
7349     \exp_not:N \int_eval:n
7350       {
7351         0
7352         \exp_not:N \__clist_count:w \c_space_tl
7353         #1 \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
7354       }
7355   }
7356 \cs_new:Npn \__clist_count:n #1 { + \c_one }
7357 \cs_new:Npx \__clist_count:w #1 ,
7358   {
7359     \exp_not:n { \exp_args:Nf \quark_if_recursion_tail_stop:n } {#1}
7360     \exp_not:N \tl_if_blank:nF {#1} { + \c_one }
7361     \exp_not:N \__clist_count:w \c_space_tl
7362   }
```

(End definition for `\clist_count:N` *and others. These functions are documented on page 128.)*

13.8 Using comma lists

<div style="margin-left: 2em;">

`\clist_use:Nnnn`
`\clist_use:cnnn`
`\__clist_use:wwn`
`\__clist_use:nwwwnwn`
`\__clist_use:nwwn`
`\clist_use:Nn`
`\clist_use:cn`

</div>

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:nwwwnwn` 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 `\q_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 `\q_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, `\q_mark` is taken as a third item, and now the second

435

`\q_mark` serves as a delimiter to `use_ii`, switching to the other ⟨*continuation*⟩, `use_iii`, which uses the ⟨*separator between final two*⟩.

```
7363 \cs_new:Npn \clist_use:Nnnn #1#2#3#4
7364   {
7365     \clist_if_exist:NTF #1
7366       {
7367         \int_case:nnF { \clist_count:N #1 }
7368           {
7369             { 0 } { }
7370             { 1 } { \exp_after:wN \__clist_use:wwn #1 , , { } }
7371             { 2 } { \exp_after:wN \__clist_use:wwn #1 , {#2} }
7372           }
7373           {
7374             \exp_after:wN \__clist_use:nwwwwnwn
7375             \exp_after:wN { \exp_after:wN } #1 ,
7376             \q_mark , { \__clist_use:nwwwwnwn {#3} }
7377             \q_mark , { \__clist_use:nwwn {#4} }
7378             \q_stop { }
7379           }
7380       }
7381       {
7382         \__msg_kernel_expandable_error:nnn
7383           { kernel } { bad-variable } {#1}
7384       }
7385   }
7386 \cs_generate_variant:Nn \clist_use:Nnnn { c }
7387 \cs_new:Npn \__clist_use:wwn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
7388 \cs_new:Npn \__clist_use:nwwwwnwn
7389     #1#2 , #3 , #4 , #5 \q_mark , #6#7 \q_stop #8
7390   { #6 {#3} , {#4} , #5 \q_mark , {#6} #7 \q_stop { #8 #1 #2 } }
7391 \cs_new:Npn \__clist_use:nwwn #1#2 , #3 \q_stop #4
7392   { \exp_not:n { #4 #1 #2 } }
7393 \cs_new:Npn \clist_use:Nn #1#2
7394   { \clist_use:Nnnn #1 {#2} {#2} {#2} }
7395 \cs_generate_variant:Nn \clist_use:Nn { c }
```

(*End definition for* `\clist_use:Nnnn` *and others. These functions are documented on page 128.*)

13.9   Using a single item

`\clist_item:Nn`
`\clist_item:cn`
`\__clist_item:nnNn`
`\__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 −⟨*length*⟩, or more than ⟨*length*⟩, the result is empty. If it is negative, but not less than −⟨*length*⟩, add ⟨*length*⟩ + 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.

```
7396 \cs_new:Npn \clist_item:Nn #1#2
7397   {
7398     \exp_args:Nfo \__clist_item:nnNn
7399       { \clist_count:N #1 }
7400       #1
7401       \__clist_item_N_loop:nw
7402       {#2}
7403   }
```

```
7404 \cs_new:Npn \__clist_item:nnNn #1#2#3#4
7405   {
7406     \int_compare:nNnTF {#4} < \c_zero
7407       {
7408         \int_compare:nNnTF {#4} < { - #1 }
7409           { \use_none_delimit_by_q_stop:w }
7410           { \exp_args:Nf #3 { \int_eval:n { #4 + \c_one + #1 } } } }
7411       }
7412       {
7413         \int_compare:nNnTF {#4} > {#1}
7414           { \use_none_delimit_by_q_stop:w }
7415           { #3 {#4} }
7416       }
7417     { } , #2 , \q_stop
7418   }
7419 \cs_new:Npn \__clist_item_N_loop:nw #1 #2,
7420   {
7421     \int_compare:nNnTF {#1} = \c_zero
7422       { \use_i_delimit_by_q_stop:nw { \exp_not:n {#2} } }
7423       { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
7424   }
7425 \cs_generate_variant:Nn \clist_item:Nn { c }
```

(*End definition for* \clist_item:Nn*,* __clist_item:nnNn*, and* __clist_item_N_loop:nw*. These functions are documented on page* *130*.)

\clist_item:nn
__clist_item_n:nw
__clist_item_n_loop:nw
__clist_item_n_end: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.

```
7426 \cs_new:Npn \clist_item:nn #1#2
7427   {
7428     \exp_args:Nf \__clist_item:nnNn
7429       { \clist_count:n {#1} }
7430       {#1}
7431       \__clist_item_n:nw
7432       {#2}
7433   }
7434 \cs_new:Npn \__clist_item_n:nw #1
7435   { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
7436 \cs_new:Npn \__clist_item_n_loop:nw #1 #2,
7437   {
7438     \exp_args:No \tl_if_blank:nTF {#2}
7439       { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
7440       {
7441         \int_compare:nNnTF {#1} = \c_zero
7442           { \exp_args:No \__clist_item_n_end:n {#2} }
7443           {
7444             \exp_args:Nf \__clist_item_n_loop:nw
7445               { \int_eval:n { #1 - 1 } }
7446             \prg_do_nothing:
7447           }
7448       }
7449   }
7450 \cs_new:Npn \__clist_item_n_end:n #1 #2 \q_stop
```

437

```
7451    {
7452      \__tl_trim_spaces:nn { \q_mark #1 }
7453        { \exp_last_unbraced:No \__clist_item_n_strip:w } ,
7454    }
7455 \cs_new:Npn \__clist_item_n_strip:w #1 , { \exp_not:n {#1} }
```

(*End definition for* `\clist_item:nn` *and others. These functions are documented on page 130.*)

13.10   Viewing comma lists

\clist_show:N
\clist_show:c
\clist_show:n

Apply the general `\__msg_show_variable:NNNnn`. In the case of an n-type comma-list, we must do things by hand, using the same message `show-clist` as for an N-type comma-list but with an empty name (first argument).

```
7456 \cs_new_protected:Npn \clist_show:N #1
7457    {
7458      \__msg_show_variable:NNNnn #1
7459        \clist_if_exist:NTF \clist_if_empty:NTF { clist }
7460        { \clist_map_function:NN #1 \__msg_show_item:n }
7461    }
7462 \cs_new_protected:Npn \clist_show:n #1
7463    {
7464      \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-clist }
7465        { } { \clist_if_empty:nF {#1} { ? } } { } { }
7466      \__msg_show_wrap:n
7467        { \clist_map_function:nN {#1} \__msg_show_item:n }
7468    }
7469 \cs_generate_variant:Nn \clist_show:N { c }
```

(*End definition for* `\clist_show:N` *and* `\clist_show:n`*. These functions are documented on page 130.*)

13.11   Scratch comma lists

\l_tmpa_clist
\l_tmpb_clist
\g_tmpa_clist
\g_tmpb_clist

Temporary comma list variables.

```
7470 \clist_new:N \l_tmpa_clist
7471 \clist_new:N \l_tmpb_clist
7472 \clist_new:N \g_tmpa_clist
7473 \clist_new:N \g_tmpb_clist
```

(*End definition for* `\l_tmpa_clist` *and others. These variables are documented on page 130.*)

```
7474 ⟨/initex | package⟩
```

14   **l3prop** implementation

The following test files are used for this code: *m3prop001, m3prop002, m3prop003,* *m3prop004, m3show001.*

```
7475 ⟨*initex | package⟩
```

```
7476 ⟨@@=prop⟩
```

A property list is a macro whose top-level expansion is of the form

$$\s__prop \ \__prop_pair:wn \ \langle key_1 \rangle \ \s__prop \ \{ \langle value_1 \rangle \}$$

$$\cdots$$

$$\__prop_pair:wn \ \langle key_n \rangle \ \s__prop \ \{ \langle value_n \rangle \}$$

where `\s__prop` is a scan mark (equal to `\scan_stop:`), and `\__prop_pair:wn` can be used to map through the 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.

```
7477 \__scan_new:N \s__prop
```

(*End definition for* `\s__prop`.)

`\__prop_pair:wn` The delimiter is always defined, but when misused simply triggers an error and removes its argument.

```
7478 \cs_new:Npn \__prop_pair:wn #1 \s__prop #2
7479   { \__msg_kernel_expandable_error:nn { kernel } { misused-prop } }
```

(*End definition for* `\__prop_pair:wn`.)

`\l__prop_internal_tl` Token list used to store the new key–value pair inserted by `\prop_put:Nnn` and friends.

```
7480 \tl_new:N \l__prop_internal_tl
```

(*End definition for* `\l__prop_internal_tl`.)

`\c_empty_prop` An empty prop.

```
7481 \tl_const:Nn \c_empty_prop { \s__prop }
```

(*End definition for* `\c_empty_prop`. *This variable is documented on page 136.*)

14.1 Allocation and initialisation

`\prop_new:N` Property lists are initialized with the value `\c_empty_prop`.
`\prop_new:c`
```
7482 \cs_new_protected:Npn \prop_new:N #1
7483   {
7484     \__chk_if_free_cs:N #1
7485     \cs_gset_eq:NN #1 \c_empty_prop
7486   }
7487 \cs_generate_variant:Nn \prop_new:N { c }
```

(*End definition for* `\prop_new:N`. *This function is documented on page 131.*)

`\prop_clear:N` The same idea for clearing.
`\prop_clear:c`
`\prop_gclear:N`
`\prop_gclear:c`
```
7488 \cs_new_protected:Npn \prop_clear:N  #1
7489   { \prop_set_eq:NN #1 \c_empty_prop }
7490 \cs_generate_variant:Nn \prop_clear:N  { c }
7491 \cs_new_protected:Npn \prop_gclear:N #1
7492   { \prop_gset_eq:NN #1 \c_empty_prop }
7493 \cs_generate_variant:Nn \prop_gclear:N { c }
```

(*End definition for* `\prop_clear:N` *and* `\prop_gclear:N`. *These functions are documented on page 131.*)

439

`\prop_clear_new:N`
`\prop_clear_new:c`
`\prop_gclear_new:N`
`\prop_gclear_new:c`

Once again a simple variation of the token list functions.

```
7494 \cs_new_protected:Npn \prop_clear_new:N #1
7495   { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
7496 \cs_generate_variant:Nn \prop_clear_new:N { c }
7497 \cs_new_protected:Npn \prop_gclear_new:N #1
7498   { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
7499 \cs_generate_variant:Nn \prop_gclear_new:N { c }
```

(*End definition for* `\prop_clear_new:N` *and* `\prop_gclear_new:N`. *These functions are documented on page 131.*)

`\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`

These are simply copies from the token list functions.

```
7500 \cs_new_eq:NN \prop_set_eq:NN   \tl_set_eq:NN
7501 \cs_new_eq:NN \prop_set_eq:Nc   \tl_set_eq:Nc
7502 \cs_new_eq:NN \prop_set_eq:cN   \tl_set_eq:cN
7503 \cs_new_eq:NN \prop_set_eq:cc   \tl_set_eq:cc
7504 \cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
7505 \cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
7506 \cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
7507 \cs_new_eq:NN \prop_gset_eq:cc \tl_gset_eq:cc
```

(*End definition for* `\prop_set_eq:NN` *and* `\prop_gset_eq:NN`. *These functions are documented on page 131.*)

`\l_tmpa_prop`
`\l_tmpb_prop`
`\g_tmpa_prop`
`\g_tmpb_prop`

We can now initialize the scratch variables.

```
7508 \prop_new:N \l_tmpa_prop
7509 \prop_new:N \l_tmpb_prop
7510 \prop_new:N \g_tmpa_prop
7511 \prop_new:N \g_tmpb_prop
```

(*End definition for* `\l_tmpa_prop` *and others. These variables are documented on page 135.*)

14.2 Accessing data in property lists

`\__prop_split:NnTF`
`\__prop_split_aux:NnTF`
`\__prop_split_aux:w`

This function is used by most of the module, and hence must be fast. It receives a ⟨*property list*⟩, a ⟨*key*⟩, a ⟨*true code*⟩ and a ⟨*false code*⟩. The aim is to split the ⟨*property list*⟩ at the given ⟨*key*⟩ into the ⟨*extract₁*⟩ before the key–value pair, the ⟨*value*⟩ associated with the ⟨*key*⟩ and the ⟨*extract₂*⟩ after the key–value pair. This is done using a delimited function, whose definition is as follows, where the ⟨*key*⟩ is turned into a string.

```
\cs_set:Npn \__prop_split_aux:w #1
\__prop_pair:wn ⟨key⟩ \s__prop #2
#3 \q_mark #4 #5 \q_stop
{ #4 {⟨true code⟩} {⟨false code⟩} }
```

If the ⟨*key*⟩ is present in the property list, `\__prop_split_aux:w`'s #1 is the part before the ⟨*key*⟩, #2 is the ⟨*value*⟩, #3 is the part after the ⟨*key*⟩, #4 is `\use_i:nn`, and #5 is additional tokens that we do not care about. The ⟨*true code*⟩ is left in the input stream, and can use the parameters #1, #2, #3 for the three parts of the property list as desired. Namely, the original property list is in this case #1 `\__prop_pair:wn` ⟨*key*⟩ `\s__prop` {#2} #3.

If the ⟨*key*⟩ is not there, then the ⟨*function*⟩ is `\use_ii:nn`, which keeps the ⟨*false code*⟩.

```
7512 \cs_new_protected:Npn \__prop_split:NnTF #1#2
```

```
7513       { \exp_args:NNo \__prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }
7514   \cs_new_protected:Npn \__prop_split_aux:NnTF #1#2#3#4
7515     {
7516       \cs_set:Npn \__prop_split_aux:w ##1
7517         \__prop_pair:wn #2 \s__prop ##2 ##3 \q_mark ##4 ##5 \q_stop
7518         { ##4 {#3} {#4} }
7519       \exp_after:wN \__prop_split_aux:w #1 \q_mark \use_i:nn
7520         \__prop_pair:wn #2 \s__prop { } \q_mark \use_ii:nn \q_stop
7521     }
7522   \cs_new:Npn \__prop_split_aux:w { }
```

(*End definition for* `\__prop_split:NnTF,` `\__prop_split_aux:NnTF,` *and* `\__prop_split_aux:w.`)

`\prop_remove:Nn`
`\prop_remove:NV`
`\prop_remove:cn`
`\prop_remove:cV`
`\prop_gremove:Nn`
`\prop_gremove:NV`
`\prop_gremove:cn`
`\prop_gremove:cV`

Deleting from a property starts by splitting the list. If the key is present in the property list, the returned value is ignored. If the key is missing, nothing happens.

```
7523   \cs_new_protected:Npn \prop_remove:Nn #1#2
7524     {
7525       \__prop_split:NnTF #1 {#2}
7526         { \tl_set:Nn #1 { ##1 ##3 } }
7527         { }
7528     }
7529   \cs_new_protected:Npn \prop_gremove:Nn #1#2
7530     {
7531       \__prop_split:NnTF #1 {#2}
7532         { \tl_gset:Nn #1 { ##1 ##3 } }
7533         { }
7534     }
7535   \cs_generate_variant:Nn \prop_remove:Nn  {      NV }
7536   \cs_generate_variant:Nn \prop_remove:Nn  { c , cV }
7537   \cs_generate_variant:Nn \prop_gremove:Nn {      NV }
7538   \cs_generate_variant:Nn \prop_gremove:Nn { c , cV }
```

(*End definition for* `\prop_remove:Nn` *and* `\prop_gremove:Nn.` *These functions are documented on page 133.*)

`\prop_get:NnN`
`\prop_get:NVN`
`\prop_get:NoN`
`\prop_get:cnN`
`\prop_get:cVN`
`\prop_get:coN`

Getting an item from a list is very easy: after splitting, if the key is in the property list, just set the token list variable to the return value, otherwise to `\q_no_value`.

```
7539   \cs_new_protected:Npn \prop_get:NnN #1#2#3
7540     {
7541       \__prop_split:NnTF #1 {#2}
7542         { \tl_set:Nn #3 {##2} }
7543         { \tl_set:Nn #3 { \q_no_value } }
7544     }
7545   \cs_generate_variant:Nn \prop_get:NnN {     NV , No }
7546   \cs_generate_variant:Nn \prop_get:NnN { c , cV , co }
```

(*End definition for* `\prop_get:NnN.` *This function is documented on page 132.*)

`\prop_pop:NnN`
`\prop_pop:NoN`
`\prop_pop:cnN`
`\prop_pop:coN`
`\prop_gpop:NnN`
`\prop_gpop:NoN`
`\prop_gpop:cnN`
`\prop_gpop:coN`

Popping a value also starts by doing the split. If the key is present, save the value in the token list and update the property list as when deleting. If the key is missing, save `\q_no_value` in the token list.

```
7547   \cs_new_protected:Npn \prop_pop:NnN #1#2#3
7548     {
7549       \__prop_split:NnTF #1 {#2}
```

441

```
7550        {
7551          \tl_set:Nn #3 {##2}
7552          \tl_set:Nn #1 { ##1 ##3 }
7553        }
7554        { \tl_set:Nn #3 { \q_no_value } }
7555    }
7556 \cs_new_protected:Npn \prop_gpop:NnN #1#2#3
7557    {
7558      \__prop_split:NnTF #1 {#2}
7559        {
7560          \tl_set:Nn #3 {##2}
7561          \tl_gset:Nn #1 { ##1 ##3 }
7562        }
7563        { \tl_set:Nn #3 { \q_no_value } }
7564    }
7565 \cs_generate_variant:Nn \prop_pop:NnN  {      No }
7566 \cs_generate_variant:Nn \prop_pop:NnN  { c , co }
7567 \cs_generate_variant:Nn \prop_gpop:NnN {      No }
7568 \cs_generate_variant:Nn \prop_gpop:NnN { c , co }
```

(*End definition for* `\prop_pop:NnN` *and* `\prop_gpop:NnN`*. These functions are documented on page* *132*.)

\prop_item:Nn
\prop_item:cn
__prop_item_Nn:nwwn

Getting the value corresponding to a key in a property list in an expandable fashion is similar to mapping some tokens. Go through the property list one ⟨*key*⟩–⟨*value*⟩ pair at a time: the arguments of `\__prop_item_Nn:nwn` are the ⟨*key*⟩ we are looking for, a ⟨*key*⟩ of the property list, and its associated value. The ⟨*keys*⟩ are compared (as strings). If they match, the ⟨*value*⟩ is returned, within `\exp_not:n`. The loop terminates even if the ⟨*key*⟩ is missing, and yields an empty value, because we have appended the appropriate ⟨*key*⟩–⟨*empty value*⟩ pair to the property list.

```
7569 \cs_new:Npn \prop_item:Nn #1#2
7570    {
7571      \exp_last_unbraced:Noo \__prop_item_Nn:nwwn { \tl_to_str:n {#2} } #1
7572        \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
7573      \__prg_break_point:
7574    }
7575 \cs_new:Npn \__prop_item_Nn:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
7576    {
7577      \str_if_eq_x:nnTF {#1} {#3}
7578        { \__prg_break:n { \exp_not:n {#4} } }
7579        { \__prop_item_Nn:nwwn {#1} }
7580    }
7581 \cs_generate_variant:Nn \prop_item:Nn { c }
```

(*End definition for* `\prop_item:Nn` *and* `\__prop_item_Nn:nwwn`*. These functions are documented on page* *133*.)

\prop_pop:NnN*TF*
\prop_pop:cnN*TF*
\prop_gpop:NnN*TF*
\prop_gpop:cnN*TF*

Popping an item from a property list, keeping track of whether the key was present or not, is implemented as a conditional. If the key was missing, neither the property list, nor the token list are altered. Otherwise, `\prg_return_true:` is used after the assignments.

```
7582 \prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
7583    {
7584      \__prop_split:NnTF #1 {#2}
7585        {
7586          \tl_set:Nn #3 {##2}
```

```
7587          \tl_set:Nn #1 { ##1 ##3 }
7588          \prg_return_true:
7589        }
7590        { \prg_return_false: }
7591    }
7592  \prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
7593    {
7594      \__prop_split:NnTF #1 {#2}
7595        {
7596          \tl_set:Nn #3 {##2}
7597          \tl_gset:Nn #1 { ##1 ##3 }
7598          \prg_return_true:
7599        }
7600        { \prg_return_false: }
7601    }
7602  \cs_generate_variant:Nn \prop_pop:NnNT   { c }
7603  \cs_generate_variant:Nn \prop_pop:NnNF   { c }
7604  \cs_generate_variant:Nn \prop_pop:NnNTF  { c }
7605  \cs_generate_variant:Nn \prop_gpop:NnNT  { c }
7606  \cs_generate_variant:Nn \prop_gpop:NnNF  { c }
7607  \cs_generate_variant:Nn \prop_gpop:NnNTF { c }
```

(*End definition for* \prop_pop:NnNTF *and* \prop_gpop:NnNTF. *These functions are documented on page* *134.*)

\prop_put:Nnn
\prop_put:NnV
\prop_put:Nno
\prop_put:Nnx
\prop_put:NVn
\prop_put:NVV
\prop_put:Non
\prop_put:Noo
\prop_put:cnn
\prop_put:cnV
\prop_put:cno
\prop_put:cnx
\prop_put:cVn
\prop_put:cVV
\prop_put:con
\prop_put:coo
\prop_gput:Nnn
\prop_gput:NnV
\prop_gput:Nno
\prop_gput:Nnx
\prop_gput:NVn
\prop_gput:NVV
\prop_gput:Non
\prop_gput:Noo
\prop_gput:cnn
\prop_gput:cnV
\prop_gput:cno
\prop_gput:cnx
\prop_gput:cVn
\prop_gput:cVV
\prop_gput:con
\prop_gput:coo
__prop_put:NNnn

Since the branches of __prop_split:NnTF are 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:NnTF. 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:NnTF. If the ⟨*key*⟩ was absent, append the new key–value to the list. Otherwise concatenate the extracts ##1 and ##3 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.

```
7608  \cs_new_protected:Npn \prop_put:Nnn  { \__prop_put:NNnn \tl_set:Nx }
7609  \cs_new_protected:Npn \prop_gput:Nnn { \__prop_put:NNnn \tl_gset:Nx }
7610  \cs_new_protected:Npn \__prop_put:NNnn #1#2#3#4
7611    {
7612      \tl_set:Nn \l__prop_internal_tl
7613        {
7614          \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
7615          \s__prop { \exp_not:n {#4} }
7616        }
7617      \__prop_split:NnTF #2 {#3}
7618        { #1 #2 { \exp_not:n {##1} \l__prop_internal_tl \exp_not:n {##3} } }
7619        { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
7620    }
7621  \cs_generate_variant:Nn \prop_put:Nnn
7622    {      NnV , Nno , Nnx , NV , NVV , No , Noo }
7623  \cs_generate_variant:Nn \prop_put:Nnn
7624    { c , cnV , cno , cnx , cV , cVV , co , coo }
7625  \cs_generate_variant:Nn \prop_gput:Nnn
7626    {      NnV , Nno , Nnx , NV , NVV , No , Noo }
7627  \cs_generate_variant:Nn \prop_gput:Nnn
7628    { c , cnV , cno , cnx , cV , cVV , co , coo }
```

443

(*End definition for* \prop_put:Nnn, \prop_gput:Nnn, *and* __prop_put:NNnn. *These functions are documented on page* *132.*)

\prop_put_if_new:Nnn
\prop_put_if_new:cnn
\prop_gput_if_new:Nnn
\prop_gput_if_new:cnn
__prop_put_if_new:NNnn

Adding conditionally also splits. If the key is already present, the three brace groups given by __prop_split:NnTF are removed. If the key is new, then the value is added, being careful to convert the key to a string using \tl_to_str:n.

```
7629 \cs_new_protected:Npn \prop_put_if_new:Nnn
7630   { \__prop_put_if_new:NNnn \tl_set:Nx }
7631 \cs_new_protected:Npn \prop_gput_if_new:Nnn
7632   { \__prop_put_if_new:NNnn \tl_gset:Nx }
7633 \cs_new_protected:Npn \__prop_put_if_new:NNnn #1#2#3#4
7634   {
7635     \tl_set:Nn \l__prop_internal_tl
7636       {
7637         \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
7638         \s__prop \exp_not:n { {#4} }
7639       }
7640     \__prop_split:NnTF #2 {#3}
7641       { }
7642       { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
7643   }
7644 \cs_generate_variant:Nn \prop_put_if_new:Nnn  { c }
7645 \cs_generate_variant:Nn \prop_gput_if_new:Nnn { c }
```

(*End definition for* \prop_put_if_new:Nnn, \prop_gput_if_new:Nnn, *and* __prop_put_if_new:NNnn. *These functions are documented on page* *132.*)

14.3 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.

```
7646 \prg_new_eq_conditional:NNn \prop_if_exist:N \cs_if_exist:N
7647   { TF , T , F , p }
7648 \prg_new_eq_conditional:NNn \prop_if_exist:c \cs_if_exist:c
7649   { TF , T , F , p }
```

(*End definition for* \prop_if_exist:NTF. *This function is documented on page* *133.*)

\prop_if_empty_p:N
\prop_if_empty_p:c
\prop_if_empty:NTF
\prop_if_empty:cTF

Same test as for token lists.

```
7650 \prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
7651   {
7652     \tl_if_eq:NNTF #1 \c_empty_prop
7653       \prg_return_true: \prg_return_false:
7654   }
7655 \cs_generate_variant:Nn \prop_if_empty_p:N { c }
7656 \cs_generate_variant:Nn \prop_if_empty:NT { c }
7657 \cs_generate_variant:Nn \prop_if_empty:NF { c }
7658 \cs_generate_variant:Nn \prop_if_empty:NTF { c }
```

(*End definition for* \prop_if_empty:NTF. *This function is documented on page* *133.*)

\prop_if_in_p:Nn
\prop_if_in_p:NV
\prop_if_in_p:No
\prop_if_in_p:cn
\prop_if_in_p:cV
\prop_if_in_p:co
\prop_if_in:NnTF
\prop_if_in:NVTF
\prop_if_in:NoTF
\prop_if_in:cnTF
\prop_if_in:cVTF
\prop_if_in:coTF
__prop_if_in:nwwn
__prop_if_in:N

Testing expandably if a key is in a property list 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:NnTF #1 {#2}
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

but `\__prop_split:NnTF` is non-expandable.

Instead, the key is compared to each key in turn using `\str_if_eq_x:nn`, which is expandable. To terminate the mapping, we append to the property list the key that is searched for. This second `\tl_to_str:n` is not expanded at the start, but only when included in the `\str_if_eq_x:nn`. It cannot make the breaking mechanism choke, because the arbitrary token list material is enclosed in braces. The second argument of `\__prop_-if_in:nwwn` is most often empty. When the ⟨*key*⟩ is found in the list, `\__prop_if_in:N` receives `\__prop_pair:wn`, and if it is found as the extra item, the function receives `\q_recursion_tail`, easily recognizable.

Here, `\prop_map_function:NN` is not sufficient for the mapping, since it can only map a single token, and cannot carry the key that is searched for.

```
7659 \prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
7660   {
7661     \exp_last_unbraced:Noo \__prop_if_in:nwwn { \tl_to_str:n {#2} } #1
7662       \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
7663       \q_recursion_tail
7664       \__prg_break_point:
7665   }
7666 \cs_new:Npn \__prop_if_in:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
7667   {
7668     \str_if_eq_x:nnTF {#1} {#3}
7669       { \__prop_if_in:N }
7670       { \__prop_if_in:nwwn {#1} }
7671   }
7672 \cs_new:Npn \__prop_if_in:N #1
7673   {
7674     \if_meaning:w \q_recursion_tail #1
7675       \prg_return_false:
7676     \else:
7677       \prg_return_true:
7678     \fi:
7679     \__prg_break:
7680   }
7681 \cs_generate_variant:Nn \prop_if_in_p:Nn {      NV , No }
7682 \cs_generate_variant:Nn \prop_if_in_p:Nn { c , cV , co }
7683 \cs_generate_variant:Nn \prop_if_in:NnT  {      NV , No }
7684 \cs_generate_variant:Nn \prop_if_in:NnT  { c , cV , co }
7685 \cs_generate_variant:Nn \prop_if_in:NnF  {      NV , No }
7686 \cs_generate_variant:Nn \prop_if_in:NnF  { c , cV , co }
7687 \cs_generate_variant:Nn \prop_if_in:NnTF {      NV , No }
7688 \cs_generate_variant:Nn \prop_if_in:NnTF { c , cV , co }
```

(*End definition for* `\prop_if_in:NnTF`, `\__prop_if_in:nwwn`, *and* `\__prop_if_in:N`. *These functions are documented on page* *133*.)

14.4 Recovering values from property lists with branching

\prop_get:NnN*TF*
\prop_get:NVN*TF*
\prop_get:NoN*TF*
\prop_get:cnN*TF*
\prop_get:cVN*TF*
\prop_get:coN*TF*

Getting the value corresponding to a key, keeping track of whether the key was present or not, is implemented as a conditional (with side effects). If the key was absent, the token list is not altered.

```
7689 \prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
7690   {
7691     \__prop_split:NnTF #1 {#2}
7692       {
7693         \tl_set:Nn #3 {##2}
7694         \prg_return_true:
7695       }
7696       { \prg_return_false: }
7697   }
7698 \cs_generate_variant:Nn \prop_get:NnNT  {      NV , No }
7699 \cs_generate_variant:Nn \prop_get:NnNF  {      NV , No }
7700 \cs_generate_variant:Nn \prop_get:NnNTF {      NV , No }
7701 \cs_generate_variant:Nn \prop_get:NnNT  { c , cV , co }
7702 \cs_generate_variant:Nn \prop_get:NnNF  { c , cV , co }
7703 \cs_generate_variant:Nn \prop_get:NnNTF { c , cV , co }
```

(*End definition for* \prop_get:NnNTF. *This function is documented on page 134.*)

14.5 Mapping to property lists

\prop_map_function:NN
\prop_map_function:Nc
\prop_map_function:cN
\prop_map_function:cc
__prop_map_function:Nwwn

The fastest way to do a recursion here is to use an \if_meaning:w test: the keys are strings, and thus cannot match the marker \q_recursion_tail. A special case to note is when the key #3 is empty: then \q_recursion_tail is compared to \exp_after:wN, also different. Note that #2 is empty, except at the first iteration, where it is \s__prop.

```
7704 \cs_new:Npn \prop_map_function:NN #1#2
7705   {
7706     \exp_last_unbraced:NNo \__prop_map_function:Nwwn #2 #1
7707       \__prop_pair:wn \q_recursion_tail \s__prop { }
7708     \__prg_break_point:Nn \prop_map_break: { }
7709   }
7710 \cs_new:Npn \__prop_map_function:Nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
7711   {
7712     \if_meaning:w \q_recursion_tail #3
7713       \exp_after:wN \prop_map_break:
7714     \fi:
7715     #1 {#3} {#4}
7716     \__prop_map_function:Nwwn #1
7717   }
7718 \cs_generate_variant:Nn \prop_map_function:NN {      Nc }
7719 \cs_generate_variant:Nn \prop_map_function:NN { c , cc }
```

(*End definition for* \prop_map_function:NN *and* __prop_map_function:Nwwn. *These functions are documented on page 134.*)

\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.

```
7720 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
7721   {
7722     \cs_gset_eq:cN
7723       { __prg_map_ \int_use:N \g__prg_map_int :wn } \__prop_pair:wn
7724     \int_gincr:N \g__prg_map_int
7725     \cs_gset:Npn \__prop_pair:wn ##1 \s__prop ##2 {#2}
7726     #1
7727     \__prg_break_point:Nn \prop_map_break:
7728       {
7729         \int_gdecr:N \g__prg_map_int
7730         \cs_gset_eq:Nc \__prop_pair:wn
7731           { __prg_map_ \int_use:N \g__prg_map_int :wn }
7732       }
7733   }
7734 \cs_generate_variant:Nn \prop_map_inline:Nn { c }
```

(*End definition for* `\prop_map_inline:Nn`. *This function is documented on page 134.*)

\prop_map_break:
\prop_map_break:n

The break statements are based on the general `\__prg_map_break:Nn`.

```
7735 \cs_new:Npn \prop_map_break:
7736   { \__prg_map_break:Nn \prop_map_break: { } }
7737 \cs_new:Npn \prop_map_break:n
7738   { \__prg_map_break:Nn \prop_map_break: }
```

(*End definition for* `\prop_map_break:` *and* `\prop_map_break:n`. *These functions are documented on page 135.*)

14.6 Viewing property lists

\prop_show:N
\prop_show:c

Apply the general `\__msg_show_variable:NNNnn`. Contrarily to sequences and comma lists, we use `\__msg_show_item:nn` to format both the key and the value for each pair.

```
7739 \cs_new_protected:Npn \prop_show:N #1
7740   {
7741     \__msg_show_variable:NNNnn #1
7742       \prop_if_exist:NTF \prop_if_empty:NTF { prop }
7743       { \prop_map_function:NN #1 \__msg_show_item:nn }
7744   }
7745 \cs_generate_variant:Nn \prop_show:N { c }
```

(*End definition for* `\prop_show:N`. *This function is documented on page 135.*)

```
7746 ⟨/initex | package⟩
```

15 l3box implementation

```
7747 ⟨*initex | package⟩
```

```
7748 ⟨@@=box⟩
```

The code in this module is very straight forward so I'm not going to comment it very extensively.

15.1 Creating and initialising boxes

The following test files are used for this code: **m3box001.lvt.**

447

`\box_new:N`
`\box_new:c`

Defining a new ⟨*box*⟩ register: remember that box 255 is not generally available.

```
7749 ⟨*package⟩
7750 \cs_new_protected:Npn \box_new:N #1
7751   {
7752     \__chk_if_free_cs:N #1
7753     \cs:w newbox \cs_end: #1
7754   }
7755 ⟨/package⟩
7756 \cs_generate_variant:Nn \box_new:N { c }
```

Clear a ⟨*box*⟩ register.

`\box_clear:N`
`\box_clear:c`
`\box_gclear:N`
`\box_gclear:c`

```
7757 \cs_new_protected:Npn \box_clear:N #1
7758   { \box_set_eq:NN  #1 \c_empty_box }
7759 \cs_new_protected:Npn \box_gclear:N #1
7760   { \box_gset_eq:NN #1 \c_empty_box }
7761 \cs_generate_variant:Nn \box_clear:N  { c }
7762 \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`

```
7763 \cs_new_protected:Npn \box_clear_new:N #1
7764   { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
7765 \cs_new_protected:Npn \box_gclear_new:N #1
7766   { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
7767 \cs_generate_variant:Nn \box_clear_new:N  { c }
7768 \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`

```
7769 \cs_new_protected:Npn \box_set_eq:NN #1#2
7770   { \tex_setbox:D #1 \tex_copy:D #2 }
7771 \cs_new_protected:Npn \box_gset_eq:NN
7772   { \tex_global:D \box_set_eq:NN }
7773 \cs_generate_variant:Nn \box_set_eq:NN  { c , Nc , cc }
7774 \cs_generate_variant:Nn \box_gset_eq:NN { c , Nc , cc }
```

`\box_set_eq_clear:NN`
`\box_set_eq_clear:cN`
`\box_set_eq_clear:Nc`
`\box_set_eq_clear:cc`
`\box_gset_eq_clear:NN`
`\box_gset_eq_clear:cN`
`\box_gset_eq_clear:Nc`
`\box_gset_eq_clear:cc`

Assigning the contents of a box to be another box. This clears the second box globally (that's how TeX does it).

```
7775 \cs_new_protected:Npn \box_set_eq_clear:NN #1#2
7776   { \tex_setbox:D #1 \tex_box:D #2 }
7777 \cs_new_protected:Npn \box_gset_eq_clear:NN
7778   { \tex_global:D  \box_set_eq_clear:NN }
7779 \cs_generate_variant:Nn \box_set_eq_clear:NN  { c , Nc , cc }
7780 \cs_generate_variant:Nn \box_gset_eq_clear:NN { c , Nc , cc }
```

`\box_if_exist_p:N`
`\box_if_exist_p:c`
`\box_if_exist:NTF`
`\box_if_exist:cTF`

Copies of the cs functions defined in l3basics.

```
7781 \prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N
7782   { TF , T , F , p }
7783 \prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c
7784   { TF , T , F , p }
```

15.2 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

```
7785 \cs_new_eq:NN \box_ht:N \tex_ht:D
7786 \cs_new_eq:NN \box_dp:N \tex_dp:D
7787 \cs_new_eq:NN \box_wd:N \tex_wd:D
7788 \cs_generate_variant:Nn \box_ht:N { c }
7789 \cs_generate_variant:Nn \box_dp:N { c }
7790 \cs_generate_variant:Nn \box_wd:N { c }
```

Measuring is easy: all primitive work. These primitives are not expandable, so the derived functions are not either.

\box_set_ht:Nn
\box_set_ht:cn
\box_set_dp:Nn
\box_set_dp:cn
\box_set_wd:Nn
\box_set_wd:cn

```
7791 \cs_new_protected:Npn \box_set_dp:Nn #1#2
7792   { \box_dp:N #1 \__dim_eval:w #2 \__dim_eval_end: }
7793 \cs_new_protected:Npn \box_set_ht:Nn #1#2
7794   { \box_ht:N #1 \__dim_eval:w #2 \__dim_eval_end: }
7795 \cs_new_protected:Npn \box_set_wd:Nn #1#2
7796   { \box_wd:N #1 \__dim_eval:w #2 \__dim_eval_end: }
7797 \cs_generate_variant:Nn \box_set_ht:Nn { c }
7798 \cs_generate_variant:Nn \box_set_dp:Nn { c }
7799 \cs_generate_variant:Nn \box_set_wd:Nn { c }
```

15.3 Using boxes

Using a ⟨*box*⟩. These are just TeX primitives with meaningful names.

\box_use_clear:N
\box_use_clear:c
\box_use:N
\box_use:c

```
7800 \cs_new_eq:NN \box_use_clear:N \tex_box:D
7801 \cs_new_eq:NN \box_use:N \tex_copy:D
7802 \cs_generate_variant:Nn \box_use_clear:N { c }
7803 \cs_generate_variant:Nn \box_use:N { c }
```

Move box material in different directions.

\box_move_left:nn
\box_move_right:nn
\box_move_up:nn
\box_move_down:nn

```
7804 \cs_new_protected:Npn \box_move_left:nn #1#2
7805   { \tex_moveleft:D \__dim_eval:w #1 \__dim_eval_end: #2 }
7806 \cs_new_protected:Npn \box_move_right:nn #1#2
7807   { \tex_moveright:D \__dim_eval:w #1 \__dim_eval_end: #2 }
7808 \cs_new_protected:Npn \box_move_up:nn #1#2
7809   { \tex_raise:D \__dim_eval:w #1 \__dim_eval_end: #2 }
7810 \cs_new_protected:Npn \box_move_down:nn #1#2
7811   { \tex_lower:D \__dim_eval:w #1 \__dim_eval_end: #2 }
```

15.4 Box conditionals

The primitives for testing if a ⟨*box*⟩ is empty/void or which type of box it is.

\if_hbox:N
\if_vbox:N
\if_box_empty:N

```
7812 \cs_new_eq:NN \if_hbox:N      \tex_ifhbox:D
7813 \cs_new_eq:NN \if_vbox:N      \tex_ifvbox:D
7814 \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*

```
7815 \prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
7816   { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
7817 \prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
7818   { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
7819 \cs_generate_variant:Nn \box_if_horizontal_p:N { c }
```

449

```
7820 \cs_generate_variant:Nn \box_if_horizontal:NT  { c }
7821 \cs_generate_variant:Nn \box_if_horizontal:NF  { c }
7822 \cs_generate_variant:Nn \box_if_horizontal:NTF { c }
7823 \cs_generate_variant:Nn \box_if_vertical_p:N { c }
7824 \cs_generate_variant:Nn \box_if_vertical:NT  { c }
7825 \cs_generate_variant:Nn \box_if_vertical:NF  { c }
7826 \cs_generate_variant:Nn \box_if_vertical:NTF { c }
```

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*

```
7827 \prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
7828   { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
7829 \cs_generate_variant:Nn \box_if_empty_p:N { c }
7830 \cs_generate_variant:Nn \box_if_empty:NT  { c }
7831 \cs_generate_variant:Nn \box_if_empty:NF  { c }
7832 \cs_generate_variant:Nn \box_if_empty:NTF { c }
```

(*End definition for* \box_new:N *and others. These functions are documented on page 137.*)

15.5   The last box inserted

\box_set_to_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c

Set a box to the previous box.

```
7833 \cs_new_protected:Npn \box_set_to_last:N #1
7834   { \tex_setbox:D #1 \tex_lastbox:D }
7835 \cs_new_protected:Npn \box_gset_to_last:N
7836   { \tex_global:D \box_set_to_last:N }
7837 \cs_generate_variant:Nn \box_set_to_last:N  { c }
7838 \cs_generate_variant:Nn \box_gset_to_last:N { c }
```

(*End definition for* \box_set_to_last:N *and* \box_gset_to_last:N*. These functions are documented on page 139.*)

15.6   Constant boxes

\c_empty_box   A box we never use.

```
7839 \box_new:N \c_empty_box
```

(*End definition for* \c_empty_box*. This variable is documented on page 140.*)

15.7   Scratch boxes

\l_tmpa_box
\l_tmpb_box
\g_tmpa_box
\g_tmpb_box

Scratch boxes.

```
7840 \box_new:N \l_tmpa_box
7841 \box_new:N \l_tmpb_box
7842 \box_new:N \g_tmpa_box
7843 \box_new:N \g_tmpb_box
```

(*End definition for* \l_tmpa_box *and others. These variables are documented on page 140.*)

15.8 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.

```
7844 \cs_new_protected:Npn \box_show:N #1
7845   { \box_show:Nnn #1 \c_max_int \c_max_int }
7846 \cs_generate_variant:Nn \box_show:N { c }
7847 \cs_new_protected:Npn \box_show:Nnn
7848   { \__box_show:NNnn \c_one }
7849 \cs_generate_variant:Nn \box_show:Nnn { c }
```

(*End definition for* `\box_show:N` *and* `\box_show:Nnn`*. These functions are documented on page 140.*)

`\box_log:N`
`\box_log:c`
`\box_log:Nnn`
`\box_log:cnn`

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.

```
7850 \cs_new_protected:Npn \box_log:N #1
7851   { \box_log:Nnn #1 \c_max_int \c_max_int }
7852 \cs_generate_variant:Nn \box_log:N { c }
7853 \cs_new_protected:Npn \box_log:Nnn #1#2#3
7854   {
7855     \use:x
7856       {
7857         \etex_interactionmode:D \c_zero
7858         \__box_show:NNnn \c_zero \exp_not:N #1
7859           { \int_eval:n {#2} } { \int_eval:n {#3} }
7860         \etex_interactionmode:D
7861             = \tex_the:D \etex_interactionmode:D \scan_stop:
7862       }
7863   }
7864 \cs_generate_variant:Nn \box_log:Nnn { c }
```

(*End definition for* `\box_log:N` *and* `\box_log:Nnn`*. These functions are documented on page 140.*)

`\__box_show:NNnn`

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.

```
7865 \cs_new_protected:Npn \__box_show:NNnn #1#2#3#4
7866   {
7867     \group_begin:
7868       \int_set:Nn \tex_showboxbreadth:D {#3}
7869       \int_set:Nn \tex_showboxdepth:D   {#4}
7870       \int_set_eq:NN \tex_tracingonline:D #1
7871       \int_set_eq:NN \tex_errorcontextlines:D \c_minus_one
7872       \box_if_exist:NTF #2
7873         { \tex_showbox:D \use:n {#2} }
7874         {
7875           \__msg_kernel_error:nnx { kernel } { variable-not-defined }
7876             { \token_to_str:N #2 }
7877         }
7878     \group_end:
7879   }
```

(*End definition for* `\__box_show:NNnn`*.*)

15.9 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.

```
7880 \cs_new_protected:Npn \hbox:n #1 { \tex_hbox:D \scan_stop: {#1} }
```

(*End definition for* \hbox:n. *This function is documented on page 140.*)

\hbox_set:Nn
\hbox_set:cn
\hbox_gset:Nn
\hbox_gset:cn

```
7881 \cs_new_protected:Npn \hbox_set:Nn #1#2
7882   { \tex_setbox:D #1 \tex_hbox:D {#2} }
7883 \cs_new_protected:Npn \hbox_gset:Nn { \tex_global:D \hbox_set:Nn }
7884 \cs_generate_variant:Nn \hbox_set:Nn { c }
7885 \cs_generate_variant:Nn \hbox_gset:Nn { c }
```

(*End definition for* \hbox_set:Nn *and* \hbox_gset:Nn. *These functions are documented on page 141.*)

\hbox_set_to_wd:Nnn   Storing material in a horizontal box with a specified width.
\hbox_set_to_wd:cnn
\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cnn

```
7886 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
7887   { \tex_setbox:D #1 \tex_hbox:D to \__dim_eval:w #2 \__dim_eval_end: {#3} }
7888 \cs_new_protected:Npn \hbox_gset_to_wd:Nnn
7889   { \tex_global:D \hbox_set_to_wd:Nnn }
7890 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
7891 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnn { c }
```

(*End definition for* \hbox_set_to_wd:Nnn *and* \hbox_gset_to_wd:Nnn. *These functions are documented on page 141.*)

\hbox_set:Nw   Storing material in a horizontal box. This type is useful in environment definitions.
\hbox_set:cw
\hbox_gset:Nw
\hbox_gset:cw
\hbox_set_end:
\hbox_gset_end:

```
7892 \cs_new_protected:Npn \hbox_set:Nw  #1
7893   { \tex_setbox:D #1 \tex_hbox:D \c_group_begin_token }
7894 \cs_new_protected:Npn \hbox_gset:Nw
7895   { \tex_global:D \hbox_set:Nw }
7896 \cs_generate_variant:Nn \hbox_set:Nw  { c }
7897 \cs_generate_variant:Nn \hbox_gset:Nw { c }
7898 \cs_new_eq:NN \hbox_set_end:  \c_group_end_token
7899 \cs_new_eq:NN \hbox_gset_end: \c_group_end_token
```

(*End definition for* \hbox_set:Nw *and others. These functions are documented on page 141.*)

\hbox_to_wd:nn   Put a horizontal box directly into the input stream.
\hbox_to_zero:n

```
7900 \cs_new_protected:Npn \hbox_to_wd:nn #1#2
7901    { \tex_hbox:D to \__dim_eval:w #1 \__dim_eval_end: {#2} }
7902 \cs_new_protected:Npn \hbox_to_zero:n #1 { \tex_hbox:D to \c_zero_dim {#1} }
```

(*End definition for* \hbox_to_wd:nn *and* \hbox_to_zero:n. *These functions are documented on page 141.*)

\hbox_overlap_left:n   Put a zero-sized box with the contents pushed against one side (which makes it stick out
\hbox_overlap_right:n   on the other) directly into the input stream.

```
7903 \cs_new_protected:Npn \hbox_overlap_left:n  #1
7904   { \hbox_to_zero:n { \tex_hss:D #1 } }
7905 \cs_new_protected:Npn \hbox_overlap_right:n #1
7906   { \hbox_to_zero:n { #1 \tex_hss:D } }
```

(*End definition for* \hbox_overlap_left:n *and* \hbox_overlap_right:n. *These functions are documented on page 141.*)

\hbox_unpack:N Unpacking a box and if requested also clear it.
\hbox_unpack:c
\hbox_unpack_clear:N
\hbox_unpack_clear:c

```
7907 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
7908 \cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D
7909 \cs_generate_variant:Nn \hbox_unpack:N { c }
7910 \cs_generate_variant:Nn \hbox_unpack_clear:N { c }
```

(*End definition for* \hbox_unpack:N *and* \hbox_unpack_clear:N. *These functions are documented on page 141.*)

15.10 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.

\vbox:n *The following test files are used for this code:* *m3box003.lvt.*

The following test files are used for this code: *m3box003.lvt.*

\vbox_top:n   Put a vertical box directly into the input stream.

```
7911 \cs_new_protected:Npn \vbox:n #1     { \tex_vbox:D { #1 \par } }
7912 \cs_new_protected:Npn \vbox_top:n #1 { \tex_vtop:D { #1 \par } }
```

(*End definition for* \vbox:n *and* \vbox_top:n. *These functions are documented on page 142.*)

\vbox_to_ht:nn Put a vertical box directly into the input stream.
\vbox_to_zero:n
\vbox_to_ht:nn
\vbox_to_zero:n

```
7913 \cs_new_protected:Npn \vbox_to_ht:nn #1#2
7914     { \tex_vbox:D to \__dim_eval:w #1 \__dim_eval_end: { #2 \par } }
7915 \cs_new_protected:Npn \vbox_to_zero:n #1
7916     { \tex_vbox:D to \c_zero_dim { #1 \par } }
```

(*End definition for* \vbox_to_ht:nn *and others. These functions are documented on page 142.*)

\vbox_set:Nn Storing material in a vertical box with a natural height.
\vbox_set:cn
\vbox_gset:Nn
\vbox_gset:cn

```
7917 \cs_new_protected:Npn \vbox_set:Nn #1#2
7918     { \tex_setbox:D #1 \tex_vbox:D { #2 \par } }
7919 \cs_new_protected:Npn \vbox_gset:Nn { \tex_global:D \vbox_set:Nn }
7920 \cs_generate_variant:Nn \vbox_set:Nn  { c }
7921 \cs_generate_variant:Nn \vbox_gset:Nn { c }
```

(*End definition for* \vbox_set:Nn *and* \vbox_gset:Nn. *These functions are documented on page 142.*)

\vbox_set_top:Nn Storing material in a vertical box with a natural height and reference point at the baseline
\vbox_set_top:cn of the first object in the box.
\vbox_gset_top:Nn
\vbox_gset_top:cn

```
7922 \cs_new_protected:Npn \vbox_set_top:Nn #1#2
7923     { \tex_setbox:D #1 \tex_vtop:D { #2 \par } }
7924 \cs_new_protected:Npn \vbox_gset_top:Nn
7925     { \tex_global:D \vbox_set_top:Nn }
7926 \cs_generate_variant:Nn \vbox_set_top:Nn { c }
7927 \cs_generate_variant:Nn \vbox_gset_top:Nn { c }
```

(*End definition for* \vbox_set_top:Nn *and* \vbox_gset_top:Nn. *These functions are documented on page 142.*)

Storing material in a vertical box with a specified height.

```
\vbox_set_to_ht:Nnn
\vbox_set_to_ht:cnn
\vbox_gset_to_ht:Nnn
\vbox_gset_to_ht:cnn
```

```
7928 \cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3
7929   {
7930     \tex_setbox:D #1 \tex_vbox:D to \__dim_eval:w #2 \__dim_eval_end:
7931       { #3 \par }
7932   }
7933 \cs_new_protected:Npn \vbox_gset_to_ht:Nnn
7934   { \tex_global:D \vbox_set_to_ht:Nnn }
7935 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn  { c }
7936 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnn { c }
```

(*End definition for* \vbox_set_to_ht:Nnn *and* \vbox_gset_to_ht:Nnn. *These functions are documented on page 142.*)

Storing material in a vertical box. This type is useful in environment definitions.

```
\vbox_set:Nw
\vbox_set:cw
\vbox_gset:Nw
\vbox_gset:cw
\vbox_set_end:
\vbox_gset_end:
```

```
7937 \cs_new_protected:Npn \vbox_set:Nw #1
7938   { \tex_setbox:D #1 \tex_vbox:D \c_group_begin_token }
7939 \cs_new_protected:Npn \vbox_gset:Nw
7940   { \tex_global:D \vbox_set:Nw }
7941 \cs_generate_variant:Nn \vbox_set:Nw  { c }
7942 \cs_generate_variant:Nn \vbox_gset:Nw { c }
7943 \cs_new_protected:Npn \vbox_set_end:
7944   {
7945     \par
7946     \c_group_end_token
7947   }
7948 \cs_new_eq:NN \vbox_gset_end: \vbox_set_end:
```

(*End definition for* \vbox_set:Nw *and others. These functions are documented on page 143.*)

Unpacking a box and if requested also clear it.

```
\vbox_unpack:N
\vbox_unpack:c
\vbox_unpack_clear:N
\vbox_unpack_clear:c
```

```
7949 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
7950 \cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D
7951 \cs_generate_variant:Nn \vbox_unpack:N { c }
7952 \cs_generate_variant:Nn \vbox_unpack_clear:N { c }
```

(*End definition for* \vbox_unpack:N *and* \vbox_unpack_clear:N. *These functions are documented on page 143.*)

Splitting a vertical box in two.

```
\vbox_set_split_to_ht:NNn
```

```
7953 \cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
7954   { \tex_setbox:D #1 \tex_vsplit:D #2 to \__dim_eval:w #3 \__dim_eval_end: }
```

(*End definition for* \vbox_set_split_to_ht:NNn. *This function is documented on page 143.*)

```
7955 ⟨/initex | package⟩
```

16 **l3coffins** Implementation

```
7956 ⟨*initex | package⟩
```

```
7957 ⟨@@=coffin⟩
```

16.1 Coffins: data structures and general variables

```
\l__coffin_internal_box
\l__coffin_internal_dim
\l__coffin_internal_tl
```
Scratch variables.

```
7958  \box_new:N \l__coffin_internal_box
7959  \dim_new:N \l__coffin_internal_dim
7960  \tl_new:N  \l__coffin_internal_tl
```

(*End 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.

```
7961  \prop_new:N \c__coffin_corners_prop
7962  \prop_put:Nnn \c__coffin_corners_prop { tl } { { 0 pt } { 0 pt } }
7963  \prop_put:Nnn \c__coffin_corners_prop { tr } { { 0 pt } { 0 pt } }
7964  \prop_put:Nnn \c__coffin_corners_prop { bl } { { 0 pt } { 0 pt } }
7965  \prop_put:Nnn \c__coffin_corners_prop { br } { { 0 pt } { 0 pt } }
```

(*End definition for* \c__coffin_corners_prop*.*)

\c__coffin_poles_prop   Pole positions are given for horizontal, vertical and reference-point based values.

```
7966  \prop_new:N \c__coffin_poles_prop
7967  \tl_set:Nn \l__coffin_internal_tl { { 0 pt } { 0 pt } { 0 pt } { 1000 pt } }
7968  \prop_put:Nno \c__coffin_poles_prop { l }  { \l__coffin_internal_tl }
7969  \prop_put:Nno \c__coffin_poles_prop { hc } { \l__coffin_internal_tl }
7970  \prop_put:Nno \c__coffin_poles_prop { r }  { \l__coffin_internal_tl }
7971  \tl_set:Nn \l__coffin_internal_tl { { 0 pt } { 0 pt } { 1000 pt } { 0 pt } }
7972  \prop_put:Nno \c__coffin_poles_prop { b }  { \l__coffin_internal_tl }
7973  \prop_put:Nno \c__coffin_poles_prop { vc } { \l__coffin_internal_tl }
7974  \prop_put:Nno \c__coffin_poles_prop { t }  { \l__coffin_internal_tl }
7975  \prop_put:Nno \c__coffin_poles_prop { B }  { \l__coffin_internal_tl }
7976  \prop_put:Nno \c__coffin_poles_prop { H }  { \l__coffin_internal_tl }
7977  \prop_put:Nno \c__coffin_poles_prop { T }  { \l__coffin_internal_tl }
```

(*End definition for* \c__coffin_poles_prop*.*)

\l__coffin_slope_x_fp   Used for calculations of intersections.
\l__coffin_slope_y_fp

```
7978  \fp_new:N \l__coffin_slope_x_fp
7979  \fp_new:N \l__coffin_slope_y_fp
```

(*End definition for* \l__coffin_slope_x_fp *and* \l__coffin_slope_y_fp*.*)

\l__coffin_error_bool   For propagating errors so that parts of the code can work around them.

```
7980  \bool_new:N \l__coffin_error_bool
```

(*End definition for* \l__coffin_error_bool*.*)

\l__coffin_offset_x_dim   The offset between two sets of coffin handles when typesetting. These values are corrected
\l__coffin_offset_y_dim   from those requested in an alignment for the positions of the handles.

```
7981  \dim_new:N \l__coffin_offset_x_dim
7982  \dim_new:N \l__coffin_offset_y_dim
```

(*End definition for* \l__coffin_offset_x_dim *and* \l__coffin_offset_y_dim*.*)

\l__coffin_pole_a_tl   Needed for finding the intersection of two poles.
\l__coffin_pole_b_tl

```
7983  \tl_new:N \l__coffin_pole_a_tl
7984  \tl_new:N \l__coffin_pole_b_tl
```

(*End definition for* \l__coffin_pole_a_tl *and* \l__coffin_pole_b_tl*.*)

For calculating intersections and so forth.

`\l__coffin_x_dim`
`\l__coffin_y_dim`
`\l__coffin_x_prime_dim`
`\l__coffin_y_prime_dim`

```
7985 \dim_new:N \l__coffin_x_dim
7986 \dim_new:N \l__coffin_y_dim
7987 \dim_new:N \l__coffin_x_prime_dim
7988 \dim_new:N \l__coffin_y_prime_dim
```

(*End definition for* `\l__coffin_x_dim` *and others.*)

16.2 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_if_exist_p:N`
`\coffin_if_exist_p:c`
`\coffin_if_exist:N`*TF*
`\coffin_if_exist:c`*TF*

Several of the higher-level coffin functions will 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.

```
7989 \prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }
7990   {
7991     \cs_if_exist:NTF #1
7992       {
7993         \cs_if_exist:cTF { l__coffin_poles_ \__int_value:w #1 _prop }
7994           { \prg_return_true: }
7995           { \prg_return_false: }
7996       }
7997       { \prg_return_false: }
7998   }
7999 \cs_generate_variant:Nn \coffin_if_exist_p:N { c }
8000 \cs_generate_variant:Nn \coffin_if_exist:NT  { c }
8001 \cs_generate_variant:Nn \coffin_if_exist:NF  { c }
8002 \cs_generate_variant:Nn \coffin_if_exist:NTF { c }
```

(*End definition for* `\coffin_if_exist:NTF`*. This function is documented on page 145.*)

`\__coffin_if_exist:NT`

Several of the higher-level coffin functions will 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.

```
8003 \cs_new_protected:Npn \__coffin_if_exist:NT #1#2
8004   {
8005     \coffin_if_exist:NTF #1
8006       { #2 }
8007       {
8008         \__msg_kernel_error:nnx { kernel } { unknown-coffin }
8009           { \token_to_str:N #1 }
8010       }
8011   }
```

(*End definition for* `\__coffin_if_exist:NT`*.*)

`\coffin_clear:N`
`\coffin_clear:c`

Clearing coffins means emptying the box and resetting all of the structures.

```
8012 \cs_new_protected:Npn \coffin_clear:N #1
8013   {
8014     \__coffin_if_exist:NT #1
8015       {
8016         \box_clear:N #1
```

```
8017                \__coffin_reset_structure:N #1
8018            }
8019    }
8020  \cs_generate_variant:Nn \coffin_clear:N { c }
```

(*End definition for* \coffin_clear:N*. This function is documented on page 145.*)

\coffin_new:N Creating a new coffin means making the underlying box and adding the data structures.
\coffin_new:c These are created globally, as there is a need to avoid any strange effects if the coffin is
created inside a group. This means that the usual rule about \l_... variables has to be
broken.

```
8021  \cs_new_protected:Npn \coffin_new:N #1
8022    {
8023      \box_new:N #1
8024      \__chk_suspend_log:
8025      \prop_clear_new:c { l__coffin_corners_ \__int_value:w #1 _prop }
8026      \prop_clear_new:c { l__coffin_poles_   \__int_value:w #1 _prop }
8027      \prop_gset_eq:cN { l__coffin_corners_ \__int_value:w #1 _prop }
8028        \c__coffin_corners_prop
8029      \prop_gset_eq:cN { l__coffin_poles_ \__int_value:w #1 _prop }
8030        \c__coffin_poles_prop
8031      \__chk_resume_log:
8032    }
8033  \cs_generate_variant:Nn \coffin_new:N { c }
```

(*End definition for* \coffin_new:N*. This function is documented on page 145.*)

\hcoffin_set:Nn Horizontal coffins are relatively easy: set the appropriate box, reset the structures then
\hcoffin_set:cn update the handle positions.

```
8034  \cs_new_protected:Npn \hcoffin_set:Nn #1#2
8035    {
8036      \__coffin_if_exist:NT #1
8037        {
8038          \hbox_set:Nn #1
8039            {
8040              \color_group_begin:
8041                \color_ensure_current:
8042                #2
8043              \color_group_end:
8044            }
8045          \__coffin_reset_structure:N #1
8046          \__coffin_update_poles:N #1
8047          \__coffin_update_corners:N #1
8048        }
8049    }
8050  \cs_generate_variant:Nn \hcoffin_set:Nn { c }
```

(*End definition for* \hcoffin_set:Nn*. This function is documented on page 145.*)

\vcoffin_set:Nnn Setting vertical coffins is more complex. First, the material is typeset with a given width.
\vcoffin_set:cnn 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).

457

```
8051 \cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
8052   {
8053     \__coffin_if_exist:NT #1
8054       {
8055         \vbox_set:Nn #1
8056           {
8057             \dim_set:Nn \tex_hsize:D {#2}
```
⟨*package⟩
8059   *\dim_set_eq:NN \linewidth   \tex_hsize:D*
8060   *\dim_set_eq:NN \columnwidth \tex_hsize:D*
8061 ⟨/package⟩
```
8062             \color_group_begin:
8063               #3
8064             \color_group_end:
8065           }
8066         \__coffin_reset_structure:N #1
8067         \__coffin_update_poles:N #1
8068         \__coffin_update_corners:N #1
8069         \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
8070         \__coffin_set_pole:Nnx #1 { T }
8071           {
8072             { 0 pt }
8073             {
8074               \dim_eval:n
8075                 { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
8076             }
8077             { 1000 pt }
8078             { 0 pt }
8079           }
8080         \box_clear:N \l__coffin_internal_box
8081       }
8082   }
8083 \cs_generate_variant:Nn \vcoffin_set:Nnn { c }
```

(*End definition for* \vcoffin_set:Nnn. *This function is documented on page 146.*)

\hcoffin_set:Nw
\hcoffin_set:cw
\hcoffin_set_end:

These are the "begin"/"end" versions of the above: watch the grouping!

```
8084 \cs_new_protected:Npn \hcoffin_set:Nw #1
8085   {
8086     \__coffin_if_exist:NT #1
8087       {
8088         \hbox_set:Nw #1 \color_group_begin: \color_ensure_current:
8089           \cs_set_protected:Npn \hcoffin_set_end:
8090             {
8091                 \color_group_end:
8092               \hbox_set_end:
8093               \__coffin_reset_structure:N #1
8094               \__coffin_update_poles:N #1
8095               \__coffin_update_corners:N #1
8096             }
8097       }
8098   }
8099 \cs_new_protected:Npn \hcoffin_set_end: { }
8100 \cs_generate_variant:Nn \hcoffin_set:Nw { c }
```

(*End definition for* \hcoffin_set:Nw *and* \hcoffin_set_end:. *These functions are documented on page 145.*)

\vcoffin_set:Nnw
\vcoffin_set:cnw
\vcoffin_set_end:

The same for vertical coffins.

```
8101 \cs_new_protected:Npn \vcoffin_set:Nnw #1#2
8102   {
8103     \__coffin_if_exist:NT #1
8104       {
8105         \vbox_set:Nw #1
8106           \dim_set:Nn \tex_hsize:D {#2}
8107 ⟨*package⟩
8108             \dim_set_eq:NN \linewidth   \tex_hsize:D
8109             \dim_set_eq:NN \columnwidth \tex_hsize:D
8110 ⟨/package⟩
8111           \color_group_begin:
8112           \cs_set_protected:Npn \vcoffin_set_end:
8113             {
8114               \color_group_end:
8115             \vbox_set_end:
8116             \__coffin_reset_structure:N #1
8117             \__coffin_update_poles:N #1
8118             \__coffin_update_corners:N #1
8119             \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
8120             \__coffin_set_pole:Nnx #1 { T }
8121               {
8122                 { 0 pt }
8123                 {
8124                   \dim_eval:n
8125                     { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
8126                 }
8127                 { 1000 pt }
8128                 { 0 pt }
8129               }
8130             \box_clear:N \l__coffin_internal_box
8131             }
8132       }
8133   }
8134 \cs_new_protected:Npn \vcoffin_set_end: { }
8135 \cs_generate_variant:Nn \vcoffin_set:Nnw { c }
```

(*End definition for* \vcoffin_set:Nnw *and* \vcoffin_set_end:. *These functions are documented on page 146.*)

\coffin_set_eq:NN
\coffin_set_eq:Nc
\coffin_set_eq:cN
\coffin_set_eq:cc

Setting two coffins equal is just a wrapper around other functions.

```
8136 \cs_new_protected:Npn \coffin_set_eq:NN #1#2
8137   {
8138     \__coffin_if_exist:NT #1
8139       {
8140         \box_set_eq:NN #1 #2
8141         \__coffin_set_eq_structure:NN #1 #2
8142       }
8143   }
8144 \cs_generate_variant:Nn \coffin_set_eq:NN { c , Nc , cc }
```

(*End definition for* \coffin_set_eq:NN. *This function is documented on page 145.*)

459

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.

```
8145 \coffin_new:N \c_empty_coffin
8146 \hbox_set:Nn  \c_empty_coffin { }
8147 \coffin_new:N \l__coffin_aligned_coffin
8148 \coffin_new:N \l__coffin_aligned_internal_coffin
```

(End definition for `\c_empty_coffin` *,* `\l__coffin_aligned_coffin` *, and* `\l__coffin_aligned_internal_-coffin`*. These variables are documented on page 148.)*

\l_tmpa_coffin The usual scratch space.
\l_tmpb_coffin
```
8149 \coffin_new:N \l_tmpa_coffin
8150 \coffin_new:N \l_tmpb_coffin
```

(End definition for `\l_tmpa_coffin` *and* `\l_tmpb_coffin`*. These variables are documented on page 148.)*

16.3 Measuring coffins

\coffin_dp:N
\coffin_dp:c
\coffin_ht:N
\coffin_ht: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.

```
8151 \cs_new_eq:NN \coffin_dp:N \box_dp:N
8152 \cs_new_eq:NN \coffin_dp:c \box_dp:c
8153 \cs_new_eq:NN \coffin_ht:N \box_ht:N
8154 \cs_new_eq:NN \coffin_ht:c \box_ht:c
8155 \cs_new_eq:NN \coffin_wd:N \box_wd:N
8156 \cs_new_eq:NN \coffin_wd:c \box_wd:c
```

(End definition for `\coffin_dp:N` *,* `\coffin_ht:N` *, and* `\coffin_wd:N`*. These functions are documented on page 147.)*

16.4 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.

```
8157 \cs_new_protected:Npn \__coffin_get_pole:NnN #1#2#3
8158   {
8159     \prop_get:cnNF
8160       { l__coffin_poles_ \__int_value:w #1 _prop } {#2} #3
8161       {
8162         \__msg_kernel_error:nnxx { kernel } { unknown-coffin-pole }
8163           {#2} { \token_to_str:N #1 }
8164         \tl_set:Nn #3 { { 0 pt } { 0 pt } { 0 pt } { 0 pt } }
8165       }
8166   }
```

(End definition for `\__coffin_get_pole:NnN`*.)*

__coffin_reset_structure:N Resetting the structure is a simple copy job.

```
8167 \cs_new_protected:Npn \__coffin_reset_structure:N #1
8168   {
8169     \prop_set_eq:cN { l__coffin_corners_ \__int_value:w #1 _prop }
8170       \c__coffin_corners_prop
8171     \prop_set_eq:cN { l__coffin_poles_ \__int_value:w #1 _prop }
```

460

```
8172          \c__coffin_poles_prop
8173       }
```

(*End definition for* `\__coffin_reset_structure:N`.)

`\__coffin_set_eq_structure:NN`
`\__coffin_gset_eq_structure:NN`

Setting coffin structures equal simply means copying the property list.

```
8174 \cs_new_protected:Npn \__coffin_set_eq_structure:NN #1#2
8175   {
8176     \prop_set_eq:cc { l__coffin_corners_ \__int_value:w #1 _prop }
8177       { l__coffin_corners_ \__int_value:w #2 _prop }
8178     \prop_set_eq:cc { l__coffin_poles_ \__int_value:w #1 _prop }
8179       { l__coffin_poles_ \__int_value:w #2 _prop }
8180   }
8181 \cs_new_protected:Npn \__coffin_gset_eq_structure:NN #1#2
8182   {
8183     \prop_gset_eq:cc { l__coffin_corners_ \__int_value:w #1 _prop }
8184       { l__coffin_corners_ \__int_value:w #2 _prop }
8185     \prop_gset_eq:cc { l__coffin_poles_ \__int_value:w #1 _prop }
8186       { l__coffin_poles_ \__int_value:w #2 _prop }
8187   }
```

(*End definition for* `\__coffin_set_eq_structure:NN` *and* `\__coffin_gset_eq_structure:NN`.)

`\coffin_set_horizontal_pole:Nnn`
`\coffin_set_horizontal_pole:cnn`
`\coffin_set_vertical_pole:Nnn`
`\coffin_set_vertical_pole:cnn`
`\__coffin_set_pole:Nnn`
`\__coffin_set_pole:Nnx`

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.

```
8188 \cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
8189   {
8190     \__coffin_if_exist:NT #1
8191       {
8192         \__coffin_set_pole:Nnx #1 {#2}
8193           {
8194             { 0 pt } { \dim_eval:n {#3} }
8195             { 1000 pt } { 0 pt }
8196           }
8197       }
8198   }
8199 \cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
8200   {
8201     \__coffin_if_exist:NT #1
8202       {
8203         \__coffin_set_pole:Nnx #1 {#2}
8204           {
8205             { \dim_eval:n {#3} } { 0 pt }
8206             { 0 pt } { 1000 pt }
8207           }
8208       }
8209   }
8210 \cs_new_protected:Npn \__coffin_set_pole:Nnn #1#2#3
8211   { \prop_put:cnn { l__coffin_poles_ \__int_value:w #1 _prop } {#2} {#3} }
8212 \cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
8213 \cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
8214 \cs_generate_variant:Nn \__coffin_set_pole:Nnn { Nnx }
```

(*End definition for* \coffin_set_horizontal_pole:Nnn, \coffin_set_vertical_pole:Nnn, *and* __- coffin_set_pole:Nnn. *These functions are documented on page* .)

__coffin_update_corners:N   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.

```
8215 \cs_new_protected:Npn \__coffin_update_corners:N #1
8216   {
8217     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { tl }
8218       { { 0 pt } { \dim_eval:n { \box_ht:N #1 } } }
8219     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { tr }
8220       { { \dim_eval:n { \box_wd:N #1 } } { \dim_eval:n { \box_ht:N #1 } } }
8221     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { bl }
8222       { { 0 pt } { \dim_eval:n { - \box_dp:N #1 } } }
8223     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { br }
8224       { { \dim_eval:n { \box_wd:N #1 } } { \dim_eval:n { -\box_dp:N #1 } } }
8225   }
```

(*End definition for* __coffin_update_corners:N.)

__coffin_update_poles:N   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.

```
8226 \cs_new_protected:Npn \__coffin_update_poles:N #1
8227   {
8228     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { hc }
8229       {
8230         { \dim_eval:n { 0.5 \box_wd:N #1 } }
8231         { 0 pt } { 0 pt } { 1000 pt }
8232       }
8233     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { r }
8234       {
8235         { \dim_eval:n { \box_wd:N #1 } }
8236         { 0 pt } { 0 pt } { 1000 pt }
8237       }
8238     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { vc }
8239       {
8240         { 0 pt }
8241         { \dim_eval:n { ( \box_ht:N #1 - \box_dp:N #1 ) / 2 } }
8242         { 1000 pt }
8243         { 0 pt }
8244       }
8245     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { t }
8246       {
8247         { 0 pt }
8248         { \dim_eval:n { \box_ht:N #1 } }
8249         { 1000 pt }
8250         { 0 pt }
8251       }
8252     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { b }
8253       {
8254         { 0 pt }
8255         { \dim_eval:n { - \box_dp:N #1 } }
8256         { 1000 pt }
```

```
8257            { 0 pt }
8258          }
8259      }
```

(*End definition for* `\__coffin_update_poles:N`.)

16.5 Coffins: calculation of pole intersections

`\__coffin_calculate_intersection:Nnn`
`\__coffin_calculate_intersection:nnnnnnnn`
`\__coffin_calculate_intersection_aux: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.

```
8260 \cs_new_protected:Npn \__coffin_calculate_intersection:Nnn #1#2#3
8261   {
8262     \__coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
8263     \__coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
8264     \bool_set_false:N \l__coffin_error_bool
8265     \exp_last_two_unbraced:Noo
8266       \__coffin_calculate_intersection:nnnnnnnn
8267         \l__coffin_pole_a_tl \l__coffin_pole_b_tl
8268     \bool_if:NT \l__coffin_error_bool
8269       {
8270         \__msg_kernel_error:nn { kernel } { no-pole-intersection }
8271         \dim_zero:N \l__coffin_x_dim
8272         \dim_zero:N \l__coffin_y_dim
8273       }
8274   }
```

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 co-ordinates 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' will be zero and a special case is needed.

```
8275 \cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnnnn
8276   #1#2#3#4#5#6#7#8
8277   {
8278     \dim_compare:nNnTF {#3} = { \c_zero_dim }
```

The case where the first pole is vertical. So the x-component of the interaction will be at a. There is then a test on the second pole: if it is also vertical then there is an error.

```
8279       {
8280         \dim_set:Nn \l__coffin_x_dim {#1}
8281         \dim_compare:nNnTF {#7} = \c_zero_dim
8282           { \bool_set_true:N \l__coffin_error_bool }
```

The second pole may still be horizontal, in which case the y-component of the intersection will be b'. If not,

$$y = \frac{d'}{c'} (x - a') + b'$$

with the x-component already known to be `#1`. This calculation is done as a generalised auxiliary.

```
8283         {
8284           \dim_compare:nNnTF {#8} = \c_zero_dim
8285             { \dim_set:Nn \l__coffin_y_dim {#6} }
```

```
8286                    {
8287                      \__coffin_calculate_intersection_aux:nnnnnN
8288                        {#1} {#5} {#6} {#7} {#8} \l__coffin_y_dim
8289                    }
8290                }
8291            }
```

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.

```
8292            {
8293              \dim_compare:nNnTF {#4} = \c_zero_dim
8294                {
8295                  \dim_set:Nn \l__coffin_y_dim {#2}
8296                  \dim_compare:nNnTF {#8} = { \c_zero_dim }
8297                    { \bool_set_true:N \l__coffin_error_bool }
8298                    {
8299                      \dim_compare:nNnTF {#7} = \c_zero_dim
8300                        { \dim_set:Nn \l__coffin_x_dim {#5} }
```

The formula for the case where the second pole is neither horizontal nor vertical is

$$x = \frac{c'}{d'} (y - b') + a'$$

which is again handled by the same auxiliary.

```
8301                        {
8302                          \__coffin_calculate_intersection_aux:nnnnnN
8303                            {#2} {#6} {#5} {#8} {#7} \l__coffin_x_dim
8304                        }
8305                    }
8306                }
```

The first pole is neither horizontal nor vertical. This still leaves the second pole, which may be a special case. For those possibilities, the calculations are the same as above with the first and second poles interchanged.

```
8307                {
8308                  \dim_compare:nNnTF {#7} = \c_zero_dim
8309                    {
8310                      \dim_set:Nn \l__coffin_x_dim {#5}
8311                      \__coffin_calculate_intersection_aux:nnnnnN
8312                        {#5} {#1} {#2} {#3} {#4} \l__coffin_y_dim
8313                    }
8314                    {
8315                      \dim_compare:nNnTF {#8} = \c_zero_dim
8316                        {
8317                          \dim_set:Nn \l__coffin_y_dim {#6}
8318                          \__coffin_calculate_intersection_aux:nnnnnN
8319                            {#6} {#2} {#1} {#4} {#3} \l__coffin_x_dim
8320                        }
```

If none of the special cases apply then there is still a need to check that there is a unique intersection between the two pole. This is the case if they have different slopes.

```
8321                        {
8322                          \fp_set:Nn \l__coffin_slope_x_fp
8323                            { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
8324                          \fp_set:Nn \l__coffin_slope_y_fp
```

```
8325                    { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
8326                  \fp_compare:nNnTF
8327                    \l__coffin_slope_x_fp = \l__coffin_slope_y_fp
8328                    { \bool_set_true:N \l__coffin_error_bool }
```

All of the tests pass, so there is the full complexity of the calculation:

$$x = \frac{a(d/c) - a'(d'/c') - b + b'}{(d/c) - (d'/c')}$$

and noting that the two ratios are already worked out from the test just performed. There is quite a bit of shuffling from dimensions to floating points in order to do the work. The y-values is then worked out using the standard auxiliary starting from the x-position.

```
8329                  {
8330                    \dim_set:Nn \l__coffin_x_dim
8331                      {
8332                        \fp_to_dim:n
8333                          {
8334                            (
8335                                  \dim_to_fp:n {#1} * \l__coffin_slope_x_fp
8336                              - ( \dim_to_fp:n {#5} * \l__coffin_slope_y_fp )
8337                              -   \dim_to_fp:n {#2}
8338                              +   \dim_to_fp:n {#6}
8339                            )
8340                            /
8341                            ( \l__coffin_slope_x_fp - \l__coffin_slope_y_fp )
8342                          }
8343                      }
8344                    \__coffin_calculate_intersection_aux:nnnnnN
8345                      { \l__coffin_x_dim }
8346                      {#5} {#6} {#8} {#7} \l__coffin_y_dim
8347                  }
8348                }
8349              }
8350            }
8351          }
8352      }
```

The formula for finding the intersection point is in most cases the same. The formula here is

$$\#6 = \#4 \cdot \left(\frac{\#1 - \#2}{\#5} \right) \#3$$

Thus #4 and #5 should be the directions of the pole while #2 and #3 are co-ordinates.

```
8353  \cs_new_protected:Npn \__coffin_calculate_intersection_aux:nnnnnN
8354      #1#2#3#4#5#6
8355    {
8356      \dim_set:Nn #6
8357        {
8358          \fp_to_dim:n
8359            {
8360              \dim_to_fp:n {#4} *
8361              ( \dim_to_fp:n {#1} - \dim_to_fp:n {#2} ) /
8362              \dim_to_fp:n {#5}
```

```
8363                    + \dim_to_fp:n {#3}
8364                  }
8365              }
8366          }
```

(*End definition for* __coffin_calculate_intersection:Nnn, __coffin_calculate_intersection:nnnnnnnn, *and* __coffin_calculate_intersection_aux:nnnnnN.)

16.6 Aligning and typesetting of coffins

\coffin_join:NnnNnnnn
\coffin_join:cnnNnnnn
\coffin_join:Nnncnnnn
\coffin_join:cnncnnnn

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 will have all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

```
8367  \cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
8368    {
8369      \__coffin_align:NnnNnnnnN
8370        #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 will 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.

```
8371      \hbox_set:Nn \l__coffin_aligned_coffin
8372        {
8373          \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
8374            { \tex_kern:D -\l__coffin_offset_x_dim }
8375          \hbox_unpack:N \l__coffin_aligned_coffin
8376          \dim_set:Nn \l__coffin_internal_dim
8377            { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
8378          \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
8379            { \tex_kern:D -\l__coffin_internal_dim }
8380        }
```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```
8381      \__coffin_reset_structure:N \l__coffin_aligned_coffin
8382      \prop_clear:c
8383        { l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _ prop }
8384      \__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 will then depend on whether any shift was needed.

```
8385      \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
8386        {
8387          \__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { 0 pt }
8388          \__coffin_offset_poles:Nnn #4 { 0 pt } { \l__coffin_offset_y_dim }
8389          \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { 0 pt }
8390          \__coffin_offset_corners:Nnn #4 { 0 pt } { \l__coffin_offset_y_dim }
8391        }
8392        {
8393          \__coffin_offset_poles:Nnn #1 { 0 pt } { 0 pt }
8394          \__coffin_offset_poles:Nnn #4
```

```
8395              { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
8396          \__coffin_offset_corners:Nnn #1 { 0 pt } { 0 pt }
8397          \__coffin_offset_corners:Nnn #4
8398              { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
8399        }
8400      \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
8401      \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
8402    }
8403  \cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cnnc }
```

(*End definition for* \coffin_join:NnnNnnnn*. This function is documented on page 147.*)

\coffin_attach:NnnNnnnn
\coffin_attach:cnnNnnnn
\coffin_attach:NnncNnnn
\coffin_attach:cnncNnnn
\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.

```
8404  \cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
8405    {
8406      \__coffin_align:NnnNnnnnN
8407        #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
8408      \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
8409      \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
8410      \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
8411      \__coffin_reset_structure:N \l__coffin_aligned_coffin
8412      \prop_set_eq:cc
8413        { l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _prop }
8414        { l__coffin_corners_ \__int_value:w #1 _prop }
8415      \__coffin_update_poles:N  \l__coffin_aligned_coffin
8416      \__coffin_offset_poles:Nnn #1 { 0 pt } { 0 pt }
8417      \__coffin_offset_poles:Nnn #4
8418        { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
8419      \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
8420      \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
8421    }
8422  \cs_new_protected:Npn \coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
8423    {
8424      \__coffin_align:NnnNnnnnN
8425        #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
8426      \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
8427      \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
8428      \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
8429      \box_set_eq:NN #1 \l__coffin_aligned_coffin
8430    }
8431  \cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnnc }
```

(*End definition for* \coffin_attach:NnnNnnnn *and* \coffin_attach_mark:NnnNnnnn*. These functions are documented on page 146.*)

__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

467

coffins. The default poles are then set up, but the final result will depend on how the bounding box is being handled.

```
8432 \cs_new_protected:Npn \__coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
8433   {
8434     \__coffin_calculate_intersection:Nnn #4 {#5} {#6}
8435     \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
8436     \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
8437     \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
8438     \dim_set:Nn \l__coffin_offset_x_dim
8439       { \l__coffin_x_dim - \l__coffin_x_prime_dim + #7 }
8440     \dim_set:Nn \l__coffin_offset_y_dim
8441       { \l__coffin_y_dim - \l__coffin_y_prime_dim + #8 }
8442     \hbox_set:Nn \l__coffin_aligned_internal_coffin
8443       {
8444         \box_use:N #1
8445         \tex_kern:D -\box_wd:N #1
8446         \tex_kern:D \l__coffin_offset_x_dim
8447         \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #4 }
8448       }
8449     \coffin_set_eq:NN #9 \l__coffin_aligned_internal_coffin
8450   }
```

(*End 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 to 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.

```
8451 \cs_new_protected:Npn \__coffin_offset_poles:Nnn #1#2#3
8452   {
8453     \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
8454       { \__coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {#2} {#3} }
8455   }
8456 \cs_new_protected:Npn \__coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
8457   {
8458     \dim_set:Nn \l__coffin_x_dim { #3 + #7 }
8459     \dim_set:Nn \l__coffin_y_dim { #4 + #8 }
8460     \tl_if_in:nnTF {#2} { - }
8461       { \tl_set:Nn \l__coffin_internal_tl { {#2} } }
8462       { \tl_set:Nn \l__coffin_internal_tl { { #1 - #2 } } }
8463     \exp_last_unbraced:NNo \__coffin_set_pole:Nnx \l__coffin_aligned_coffin
8464       { \l__coffin_internal_tl }
8465       {
8466         { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
8467         {#5} {#6}
8468       }
8469   }
```

(*End 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.

```
8470 \cs_new_protected:Npn \__coffin_offset_corners:Nnn #1#2#3
8471   {
8472     \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
8473       { \__coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
8474   }
8475 \cs_new_protected:Npn \__coffin_offset_corner:Nnnnn #1#2#3#4#5#6
8476   {
8477     \prop_put:cnx
8478       { l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _prop }
8479       { #1 - #2 }
8480       {
8481         { \dim_eval:n { #3 + #5 } }
8482         { \dim_eval:n { #4 + #6 } }
8483       }
8484   }
```

(*End 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 will 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.

```
8485 \cs_new_protected:Npn \__coffin_update_vertical_poles:NNN #1#2#3
8486   {
8487     \__coffin_get_pole:NnN #3 { #1 -T } \l__coffin_pole_a_tl
8488     \__coffin_get_pole:NnN #3 { #2 -T } \l__coffin_pole_b_tl
8489     \exp_last_two_unbraced:Noo \__coffin_update_T:nnnnnnnnN
8490       \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
8491     \__coffin_get_pole:NnN #3 { #1 -B } \l__coffin_pole_a_tl
8492     \__coffin_get_pole:NnN #3 { #2 -B } \l__coffin_pole_b_tl
8493     \exp_last_two_unbraced:Noo \__coffin_update_B:nnnnnnnnN
8494       \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
8495   }
8496 \cs_new_protected:Npn \__coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
8497   {
8498     \dim_compare:nNnTF {#2} < {#6}
8499       {
8500         \__coffin_set_pole:Nnx #9 { T }
8501           { { 0 pt } {#6} { 1000 pt } { 0 pt } }
8502       }
8503       {
8504         \__coffin_set_pole:Nnx #9 { T }
8505           { { 0 pt } {#2} { 1000 pt } { 0 pt } }
8506       }
8507   }
8508 \cs_new_protected:Npn \__coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
8509   {
8510     \dim_compare:nNnTF {#2} < {#6}
8511       {
8512         \__coffin_set_pole:Nnx #9 { B }
8513           { { 0 pt } {#2}  { 1000 pt } { 0 pt } }
8514       }
8515       {
```

```
8516          \__coffin_set_pole:Nnx #9 { B }
8517            { { 0 pt } {#6} { 1000 pt } { 0 pt } }
8518        }
8519    }
```

(End definition for __coffin_update_vertical_poles:NNN *,* __coffin_update_T:nnnnnnnnN *, and* __- *coffin_update_B:nnnnnnnnN.)*

\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.

```
8520 \cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
8521    {
8522      \hbox_unpack:N \c_empty_box
8523      \__coffin_align:NnnNnnnnN \c_empty_coffin { H } { l }
8524        #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
8525      \box_use:N \l__coffin_aligned_coffin
8526    }
8527 \cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }
```

(End definition for \coffin_typeset:Nnnnn*. This function is documented on page 147.)*

16.7 Coffin diagnostics

\l__coffin_display_coffin
\l__coffin_display_coord_coffin
\l__coffin_display_pole_coffin

Used for printing coffins with data structures attached.

```
8528 \coffin_new:N \l__coffin_display_coffin
8529 \coffin_new:N \l__coffin_display_coord_coffin
8530 \coffin_new:N \l__coffin_display_pole_coffin
```

(End 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.

```
8531 \prop_new:N \l__coffin_display_handles_prop
8532 \prop_put:Nnn \l__coffin_display_handles_prop { tl }
8533    { { b } { r } { -1 } { 1 } }
8534 \prop_put:Nnn \l__coffin_display_handles_prop { thc }
8535    { { b } { hc } { 0 } { 1 } }
8536 \prop_put:Nnn \l__coffin_display_handles_prop { tr }
8537    { { b } { l } { 1 } { 1 } }
8538 \prop_put:Nnn \l__coffin_display_handles_prop { vcl }
8539    { { vc } { r } { -1 } { 0 } }
8540 \prop_put:Nnn \l__coffin_display_handles_prop { vchc }
8541    { { vc } { hc } { 0 } { 0 } }
8542 \prop_put:Nnn \l__coffin_display_handles_prop { vcr }
8543    { { vc } { l } { 1 } { 0 } }
8544 \prop_put:Nnn \l__coffin_display_handles_prop { bl }
8545    { { t } { r } { -1 } { -1 } }
8546 \prop_put:Nnn \l__coffin_display_handles_prop { bhc }
8547    { { t } { hc } { 0 } { -1 } }
8548 \prop_put:Nnn \l__coffin_display_handles_prop { br }
8549    { { t } { l } { 1 } { -1 } }
8550 \prop_put:Nnn \l__coffin_display_handles_prop { Tl }
```

```
8551        { { t } { r } { -1 } { -1 } }
8552   \prop_put:Nnn \l__coffin_display_handles_prop { Thc }
8553        { { t } { hc } { 0 } { -1 } }
8554   \prop_put:Nnn \l__coffin_display_handles_prop { Tr }
8555        { { t } { l } { 1 } { -1 } }
8556   \prop_put:Nnn \l__coffin_display_handles_prop { Hl }
8557        { { vc } { r } { -1 } { 1 } }
8558   \prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
8559        { { vc } { hc } { 0 } { 1 } }
8560   \prop_put:Nnn \l__coffin_display_handles_prop { Hr }
8561        { { vc } { l } { 1 } { 1 } }
8562   \prop_put:Nnn \l__coffin_display_handles_prop { Bl }
8563        { { b } { r } { -1 } { -1 } }
8564   \prop_put:Nnn \l__coffin_display_handles_prop { Bhc }
8565        { { b } { hc } { 0 } { -1 } }
8566   \prop_put:Nnn \l__coffin_display_handles_prop { Br }
8567        { { b } { l } { 1 } { -1 } }
```

(*End 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.

```
8568   \dim_new:N  \l__coffin_display_offset_dim
8569   \dim_set:Nn \l__coffin_display_offset_dim { 2 pt }
```

(*End definition for* \l__coffin_display_offset_dim.)

\l__coffin_display_x_dim   As the intersections of poles have to be calculated to find which ones to print, there is
\l__coffin_display_y_dim   a need to avoid repetition. This is done by saving the intersection into two dedicated
values.

```
8570   \dim_new:N \l__coffin_display_x_dim
8571   \dim_new:N \l__coffin_display_y_dim
```

(*End 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.

```
8572   \prop_new:N \l__coffin_display_poles_prop
```

(*End definition for* \l__coffin_display_poles_prop.)

\l__coffin_display_font_tl   Stores the settings used to print coffin data: this keeps things flexible.

```
8573   \tl_new:N  \l__coffin_display_font_tl
8574 ⟨*initex⟩
8575   \tl_set:Nn \l__coffin_display_font_tl { } % TODO
8576 ⟨/initex⟩
8577 ⟨*package⟩
8578   \tl_set:Nn \l__coffin_display_font_tl { \sffamily \tiny }
8579 ⟨/package⟩
```

(*End definition for* \l__coffin_display_font_tl.)

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.

```
8580 \cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
8581   {
8582     \hcoffin_set:Nn \l__coffin_display_pole_coffin
8583       {
8584 ⟨*initex⟩
8585         \hbox:n { \tex_vrule:D width 1 pt height 1 pt \scan_stop: } % TODO
8586 ⟨/initex⟩
8587 ⟨*package⟩
8588         \color {#4}
8589         \rule { 1 pt } { 1 pt }
8590 ⟨/package⟩
8591       }
8592     \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
8593       \l__coffin_display_pole_coffin { hc } { vc } { 0 pt } { 0 pt }
8594     \hcoffin_set:Nn \l__coffin_display_coord_coffin
8595       {
8596 ⟨*initex⟩
8597         % TODO
8598 ⟨/initex⟩
8599 ⟨*package⟩
8600         \color {#4}
8601 ⟨/package⟩
8602         \l__coffin_display_font_tl
8603         ( \tl_to_str:n { #2 , #3 } )
8604       }
8605     \prop_get:NnN \l__coffin_display_handles_prop
8606       { #2 #3 } \l__coffin_internal_tl
8607     \quark_if_no_value:NTF \l__coffin_internal_tl
8608       {
8609         \prop_get:NnN \l__coffin_display_handles_prop
8610           { #3 #2 } \l__coffin_internal_tl
8611         \quark_if_no_value:NTF \l__coffin_internal_tl
8612           {
8613             \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
8614               \l__coffin_display_coord_coffin { l } { vc }
8615                 { 1 pt } { 0 pt }
8616           }
8617           {
8618             \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
8619               \l__coffin_internal_tl #1 {#2} {#3}
8620           }
8621       }
8622       {
8623         \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
8624           \l__coffin_internal_tl #1 {#2} {#3}
8625       }
8626   }
8627 \cs_new_protected:Npn \__coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
8628   {
8629     \coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
```

472

```
8630        \l__coffin_display_coord_coffin {#1} {#2}
8631          { #3 \l__coffin_display_offset_dim }
8632          { #4 \l__coffin_display_offset_dim }
8633      }
8634    \cs_generate_variant:Nn \coffin_mark_handle:Nnnn { c }
```

(*End definition for* \coffin_mark_handle:Nnnn *and* __coffin_mark_handle_aux:nnnnNnn. *These functions are documented on page 148.*)

\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.

```
8635    \cs_new_protected:Npn \coffin_display_handles:Nn #1#2
8636      {
8637        \hcoffin_set:Nn \l__coffin_display_pole_coffin
8638          {
8639 ⟨*initex⟩
8640            \hbox:n { \tex_vrule:D width 1 pt height 1 pt \scan_stop: } % TODO
8641 ⟨/initex⟩
8642 ⟨*package⟩
8643            \color {#2}
8644            \rule { 1 pt } { 1 pt }
8645 ⟨/package⟩
8646          }
8647        \prop_set_eq:Nc \l__coffin_display_poles_prop
8648          { l__coffin_poles_ \__int_value:w #1 _prop }
8649        \__coffin_get_pole:NnN #1 { H } \l__coffin_pole_a_tl
8650        \__coffin_get_pole:NnN #1 { T } \l__coffin_pole_b_tl
8651        \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
8652          { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
8653        \__coffin_get_pole:NnN #1 { B } \l__coffin_pole_b_tl
8654        \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
8655          { \prop_remove:Nn \l__coffin_display_poles_prop { B } }
8656        \coffin_set_eq:NN \l__coffin_display_coffin #1
8657        \prop_map_inline:Nn \l__coffin_display_poles_prop
8658          {
8659            \prop_remove:Nn \l__coffin_display_poles_prop {##1}
8660            \__coffin_display_handles_aux:nnnnnn {##1} ##2 {#2}
8661          }
8662        \box_use:N \l__coffin_display_coffin
8663      }
```

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.

```
8664    \cs_new_protected:Npn \__coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
8665      {
8666        \prop_map_inline:Nn \l__coffin_display_poles_prop
8667          {
8668            \bool_set_false:N \l__coffin_error_bool
8669            \__coffin_calculate_intersection:nnnnnnnn {#2} {#3} {#4} {#5} ##2
8670            \bool_if:NF \l__coffin_error_bool
8671              {
```

473

```
8672              \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
8673              \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
8674              \__coffin_display_attach:Nnnnn
8675                \l__coffin_display_pole_coffin { hc } { vc }
8676                { 0 pt } { 0 pt }
8677              \hcoffin_set:Nn \l__coffin_display_coord_coffin
8678                {
```
⟨*initex⟩
```
8680                   % TODO
```
⟨/initex⟩
⟨*package⟩
```
8683                   \color {#6}
```
⟨/package⟩
```
8685                   \l__coffin_display_font_tl
8686                   ( \tl_to_str:n { #1 , ##1 } )
8687                }
8688              \prop_get:NnN \l__coffin_display_handles_prop
8689                { #1 ##1 } \l__coffin_internal_tl
8690              \quark_if_no_value:NTF \l__coffin_internal_tl
8691                {
8692                  \prop_get:NnN \l__coffin_display_handles_prop
8693                    { ##1 #1 } \l__coffin_internal_tl
8694                  \quark_if_no_value:NTF \l__coffin_internal_tl
8695                    {
8696                      \__coffin_display_attach:Nnnnn
8697                        \l__coffin_display_coord_coffin { l } { vc }
8698                        { 1 pt } { 0 pt }
8699                    }
8700                    {
8701                      \exp_last_unbraced:No
8702                        \__coffin_display_handles_aux:nnnn
8703                        \l__coffin_internal_tl
8704                    }
8705                }
8706                {
8707                  \exp_last_unbraced:No \__coffin_display_handles_aux:nnnn
8708                    \l__coffin_internal_tl
8709                }
8710            }
8711          }
8712      }
8713 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
8714   {
8715     \__coffin_display_attach:Nnnnn
8716       \l__coffin_display_coord_coffin {#1} {#2}
8717       { #3 \l__coffin_display_offset_dim }
8718       { #4 \l__coffin_display_offset_dim }
8719   }
8720 \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.

```
8721 \cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5
```

```
8722    {
8723      \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
8724      \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
8725      \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
8726      \dim_set:Nn \l__coffin_offset_x_dim
8727        { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
8728      \dim_set:Nn \l__coffin_offset_y_dim
8729        { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
8730      \hbox_set:Nn \l__coffin_aligned_coffin
8731        {
8732          \box_use:N \l__coffin_display_coffin
8733          \tex_kern:D -\box_wd:N \l__coffin_display_coffin
8734          \tex_kern:D \l__coffin_offset_x_dim
8735          \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
8736        }
8737      \box_set_ht:Nn \l__coffin_aligned_coffin
8738        { \box_ht:N \l__coffin_display_coffin }
8739      \box_set_dp:Nn \l__coffin_aligned_coffin
8740        { \box_dp:N \l__coffin_display_coffin }
8741      \box_set_wd:Nn \l__coffin_aligned_coffin
8742        { \box_wd:N \l__coffin_display_coffin }
8743      \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
8744    }
```

(*End definition for* \coffin_display_handles:Nn *and others. These functions are documented on page* *147.*)

\coffin_show_structure:N
\coffin_show_structure:c

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.

```
8745 \cs_new_protected:Npn \coffin_show_structure:N #1
8746   {
8747     \__coffin_if_exist:NT #1
8748       {
8749         \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-coffin }
8750           { \token_to_str:N #1 }
8751           { \dim_eval:n { \coffin_ht:N #1 } }
8752           { \dim_eval:n { \coffin_dp:N #1 } }
8753           { \dim_eval:n { \coffin_wd:N #1 } }
8754         \__msg_show_wrap:n
8755           {
8756             \prop_map_function:cN
8757               { l__coffin_poles_ \__int_value:w #1 _prop }
8758               \__msg_show_item_unbraced:nn
8759           }
8760       }
8761   }
8762 \cs_generate_variant:Nn \coffin_show_structure:N { c }
```

(*End definition for* \coffin_show_structure:N*. This function is documented on page* *148.*)

16.8 Messages

```
8763 \__msg_kernel_new:nnnn { kernel } { no-pole-intersection }
8764   { No~intersection~between~coffin~poles. }
8765   {
```

```
8766        \c__msg_coding_error_text_tl
8767        LaTeX~was~asked~to~find~the~intersection~between~two~poles,~
8768        but~they~do~not~have~a~unique~meeting~point:~
8769        the~value~(0~pt,~0~pt)~will~be~used.
8770      }
8771  \__msg_kernel_new:nnnn { kernel } { unknown-coffin }
8772    { Unknown~coffin~'#1'. }
8773    { The~coffin~'#1'~was~never~defined. }
8774  \__msg_kernel_new:nnnn { kernel } { unknown-coffin-pole }
8775    { Pole~'#1'~unknown~for~coffin~'#2'. }
8776    {
8777        \c__msg_coding_error_text_tl
8778        LaTeX~was~asked~to~find~a~typesetting~pole~for~a~coffin,~
8779        but~either~the~coffin~does~not~exist~or~the~pole~name~is~wrong.
8780    }
8781  \__msg_kernel_new:nnn { kernel } { show-coffin }
8782    {
8783      Size~of~coffin~#1 : \\
8784      > ~ ht~=~#2 \\
8785      > ~ dp~=~#3 \\
8786      > ~ wd~=~#4 \\
8787      Poles~of~coffin~#1 :
8788    }
8789  ⟨/initex | package⟩
```

17   l3color Implementation

```
8790  ⟨*initex | package⟩
```

\color_group_begin: Grouping for color is almost the same as using the basic `\group_begin:` and `\group_-`
\color_group_end: end: functions. However, in vertical mode the end-of-group needs a `\par`, which in horizontal mode does nothing.

```
8791  \cs_new_eq:NN \color_group_begin: \group_begin:
8792  \cs_new_protected:Npn \color_group_end:
8793    {
8794        \tex_par:D
8795      \group_end:
8796    }
```

(*End definition for* `\color_group_begin:` *and* `\color_group_end:`. *These functions are documented on page 149.*)

\color_ensure_current: A driver-independent wrapper for setting the foreground color to the current color "now".

```
8797  ⟨*initex⟩
8798  \cs_new_protected:Npn \color_ensure_current:
8799    { \__driver_color_ensure_current: }
8800  ⟨/initex⟩
```

In package mode, the driver code may not be loaded. To keep down dependencies, if there is no driver code available and no `\set@color` then color is not in use and this function can be a no-op.

```
8801  ⟨*package⟩
8802  \cs_new_protected:Npn \color_ensure_current: { }
8803  \AtBeginDocument
```

```
8804    {
8805      \cs_if_exist:NTF \__driver_color_ensure_current:
8806        {
8807          \cs_set_protected:Npn \color_ensure_current:
8808            { \__driver_color_ensure_current: }
8809        }
8810        {
8811          \cs_if_exist:NT \set@color
8812            {
8813              \cs_set_protected:Npn \color_ensure_current:
8814                { \set@color }
8815            }
8816        }
8817    }
8818 ⟨/package⟩
```

(*End definition for* \color_ensure_current:. *This function is documented on page 149.*)

```
8819 ⟨/initex | package⟩
```

18   **l3msg** implementation

```
8820 ⟨*initex | package⟩
```

```
8821 ⟨@@=msg⟩
```

\l__msg_internal_tl   A general scratch for the module.

```
8822 \tl_new:N \l__msg_internal_tl
```

(*End definition for* \l__msg_internal_tl.)

18.1   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   Locations for the text of messages.
\c__msg_more_text_prefix_tl
```
8823 \tl_const:Nn \c__msg_text_prefix_tl      { msg~text~>~ }
8824 \tl_const:Nn \c__msg_more_text_prefix_tl { msg~extra~text~>~ }
```

(*End definition for* \c__msg_text_prefix_tl *and* \c__msg_more_text_prefix_tl.)

\msg_if_exist_p:nn   Test whether the control sequence containing the message text exists or not.
\msg_if_exist:nn*TF*
```
8825 \prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
8826   {
8827     \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
8828       { \prg_return_true: } { \prg_return_false: }
8829   }
```

(*End definition for* \msg_if_exist:nnTF. *This function is documented on page 151.*)

__chk_if_free_msg:nn This auxiliary is similar to __chk_if_free_cs:N, and is used when defining messages with \msg_new:nnnn. It could be inlined in \msg_new:nnnn, but the experimental l3trace module needs to disable this check when reloading a package with the extra tracing information.

```
8830 \cs_new_protected:Npn \__chk_if_free_msg:nn #1#2
8831   {
8832     \msg_if_exist:nnT {#1} {#2}
8833       {
8834         \__msg_kernel_error:nnxx { kernel } { message-already-defined }
8835           {#1} {#2}
8836       }
8837   }
8838 ⟨*package⟩
8839 \if_bool:N \l@expl@log@functions@bool
8840   \cs_gset_protected:Npn \__chk_if_free_msg:nn #1#2
8841     {
8842       \msg_if_exist:nnT {#1} {#2}
8843         {
8844           \__msg_kernel_error:nnxx { kernel } { message-already-defined }
8845             {#1} {#2}
8846         }
8847       \__chk_log:x { Defining~message~ #1 / #2 ~\msg_line_context: }
8848     }
8849 \fi:
8850 ⟨/package⟩
```

(*End definition for* __chk_if_free_msg:nn.)

\msg_new:nnnn Setting a message simply means saving the appropriate text into two functions. A sanity
\msg_new:nnn check first.
\msg_gset:nnnn
\msg_gset:nnn
\msg_set:nnnn
\msg_set:nnn

```
8851 \cs_new_protected:Npn \msg_new:nnnn #1#2
8852   {
8853     \__chk_if_free_msg:nn {#1} {#2}
8854     \msg_gset:nnnn {#1} {#2}
8855   }
8856 \cs_new_protected:Npn \msg_new:nnn #1#2#3
8857   { \msg_new:nnnn {#1} {#2} {#3} { } }
8858 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
8859   {
8860     \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
8861       ##1##2##3##4 {#3}
8862     \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
8863       ##1##2##3##4 {#4}
8864   }
8865 \cs_new_protected:Npn \msg_set:nnn #1#2#3
8866   { \msg_set:nnnn {#1} {#2} {#3} { } }
8867 \cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
8868   {
8869     \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
8870       ##1##2##3##4 {#3}
8871     \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
8872       ##1##2##3##4 {#4}
8873   }
8874 \cs_new_protected:Npn \msg_gset:nnn #1#2#3
```

478

```
8875        { \msg_gset:nnnn {#1} {#2} {#3} { } }
```

(*End definition for* `\msg_new:nnnn` *and others. These functions are documented on page 150.*)

18.2   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.

```
8876 \tl_const:Nn \c__msg_coding_error_text_tl
8877   {
8878     This~is~a~coding~error.
8879     \\ \\
8880   }
8881 \tl_const:Nn \c__msg_continue_text_tl
8882   { Type~<return>~to~continue }
8883 \tl_const:Nn \c__msg_critical_text_tl
8884   { Reading~the~current~file~'\g_file_current_name_tl'~will~stop. }
8885 \tl_const:Nn \c__msg_fatal_text_tl
8886   { This~is~a~fatal~error:~LaTeX~will~abort. }
8887 \tl_const:Nn \c__msg_help_text_tl
8888   { For~immediate~help~type~H~<return> }
8889 \tl_const:Nn \c__msg_no_info_text_tl
8890   {
8891     LaTeX~does~not~know~anything~more~about~this~error,~sorry.
8892     \c__msg_return_text_tl
8893   }
8894 \tl_const:Nn \c__msg_on_line_text_tl { on~line }
8895 \tl_const:Nn \c__msg_return_text_tl
8896   {
8897     \\ \\
8898     Try~typing~<return>~to~proceed.
8899     \\
8900     If~that~doesn't~work,~type~X~<return>~to~quit.
8901   }
8902 \tl_const:Nn \c__msg_trouble_text_tl
8903   {
8904     \\ \\
8905     More~errors~will~almost~certainly~follow: \\
8906     the~LaTeX~run~should~be~aborted.
8907   }
```

(*End 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`.

```
8908 \cs_new:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
8909 \cs_gset:Npn \msg_line_context:
8910   {
8911     \c__msg_on_line_text_tl
8912     \c_space_tl
8913     \msg_line_number:
8914   }
```

(*End definition for* `\msg_line_number:` *and* `\msg_line_context:`. *These functions are documented on page 151.*)

479

18.3 Showing messages: low level mechanism

`\msg_interrupt:nnn` 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`.

```
8915 \cs_new_protected:Npn \msg_interrupt:nnn #1#2#3
8916   {
8917     \tl_if_empty:nTF {#3}
8918       {
8919         \__msg_interrupt_wrap:nn { \\ \c__msg_no_info_text_tl }
8920           {#1 \\\\ #2 \\\\ \c__msg_continue_text_tl }
8921       }
8922       {
8923         \__msg_interrupt_wrap:nn { \\ #3 }
8924           {#1 \\\\ #2 \\\\ \c__msg_help_text_tl }
8925       }
8926   }
```

(*End definition for* `\msg_interrupt:nnn`. *This function is documented on page 155.*)

`\__msg_interrupt_wrap:nn`
`\__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 x-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.

```
8927 \cs_new_protected:Npn \__msg_interrupt_wrap:nn #1#2
8928   {
8929     \iow_wrap:nnnN {#1} { | ~ } { } \__msg_interrupt_more_text:n
8930     \iow_wrap:nnnN {#2} { ! ~ } { } \__msg_interrupt_text:n
8931   }
8932 \cs_new_protected:Npn \__msg_interrupt_more_text:n #1
8933   {
8934     \exp_args:Nx \tex_errhelp:D
8935       {
8936         |'''''''''''''''''''''''''''''''''''''''''''''''''
8937         #1 \iow_newline:
8938         |...............................................
8939       }
8940   }
```

(*End definition for* `\__msg_interrupt_wrap:nn` *and* `\__msg_interrupt_more_text:n`.)

`\__msg_interrupt_text: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 {⟨dots⟩}` which fills the output with dots. 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 will be inserted after the message is entirely cleaned up.

The `\__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.

```
8941 \group_begin:
8942   \char_set_lccode:nn {'\{} {'\ }
8943   \char_set_lccode:nn {'\}} {'\ }
8944   \char_set_lccode:nn {'\&} {'\!}
8945   \char_set_catcode_active:N \&
8946 \tex_lowercase:D
8947   {
8948     \group_end:
8949     \cs_new_protected:Npn \__msg_interrupt_text:n #1
8950       {
8951         \iow_term:x
8952           {
8953             \iow_newline:
8954             !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
8955             \iow_newline:
8956             !
8957           }
8958         \__iow_with:Nnn \tex_newlinechar:D { '\^^J }
8959           {
8960             \__iow_with:Nnn \tex_errorcontextlines:D \c_minus_one
8961               {
8962                 \group_begin:
8963                   \cs_set_protected:Npn &
8964                     {
8965                       \tex_errmessage:D
8966                         {
8967                           #1
8968                           \use_none:n
8969                             { ......................................... }
8970                         }
8971                     }
8972                   \exp_after:wN
8973                 \group_end:
8974                 &
8975               }
8976           }
8977       }
8978   }
```

(*End definition for* `\__msg_interrupt_text:n`.)

\msg_log:n Printing to the log or terminal without a stop is rather easier. A bit of simple visual
\msg_term:n work sets things off nicely.

```
8979 \cs_new_protected:Npn \msg_log:n #1
8980   {
8981     \iow_log:n { ............................................. }
8982     \iow_wrap:nnnN { . ~ #1} { . ~ } { } \iow_log:n
8983     \iow_log:n { ............................................. }
```

```
8984        }
8985    \cs_new_protected:Npn \msg_term:n #1
8986      {
8987        \iow_term:n { ********************************************* }
8988        \iow_wrap:nnnN { * ~ #1} { * ~ } { } \iow_term:n
8989        \iow_term:n { ********************************************* }
8990      }
```

(*End definition for* `\msg_log:n` *and* `\msg_term:n`. *These functions are documented on page* *156.*)

18.4   Displaying messages

LaTeX is handling error messages and so the TeX ones are disabled. This is already done by the LaTeX 2_ε kernel, so to avoid messing up any deliberate change by a user this is only set in format mode.

```
8991  ⟨*initex⟩
8992  \int_gset_eq:NN \tex_errorcontextlines:D \c_minus_one
8993  ⟨/initex⟩
```

`\msg_fatal_text:n`
`\msg_critical_text:n`
`\msg_error_text:n`
`\msg_warning_text:n`
`\msg_info_text:n`

A function for issuing messages: both the text and order could in principle vary.

```
8994  \cs_new:Npn \msg_fatal_text:n #1 { Fatal~#1~error }
8995  \cs_new:Npn \msg_critical_text:n #1 { Critical~#1~error }
8996  \cs_new:Npn \msg_error_text:n #1 { #1~error }
8997  \cs_new:Npn \msg_warning_text:n #1 { #1~warning }
8998  \cs_new:Npn \msg_info_text:n #1 { #1~info }
```

(*End definition for* `\msg_fatal_text:n` *and others. These functions are documented on page* *151.*)

`\msg_see_documentation_text:n`

Contextual footer information. The LaTeX module only comprises LaTeX3 code, so we refer to the LaTeX3 documentation rather than simply "LaTeX".

```
8999  \cs_new:Npn \msg_see_documentation_text:n #1
9000    {
9001      \\ \\ See~the~
9002      \str_if_eq:nnTF {#1} { LaTeX } { LaTeX3 } {#1} ~
9003      documentation~for~further~information.
9004    }
```

(*End definition for* `\msg_see_documentation_text:n`. *This function is documented on page* *152.*)

`\__msg_class_new:nn`

```
9005  \group_begin:
9006    \cs_set_protected:Npn \__msg_class_new:nn #1#2
9007      {
9008        \prop_new:c { l__msg_redirect_ #1 _prop }
9009        \cs_new_protected:cpn { __msg_ #1 _code:nnnnnn }
9010            ##1##2##3##4##5##6 {#2}
9011        \cs_new_protected:cpn { msg_ #1 :nnnnnn } ##1##2##3##4##5##6
9012          {
9013            \use:x
9014              {
9015                \exp_not:n { \__msg_use:nnnnnnn {#1} {##1} {##2} }
9016                  { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
9017                  { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
9018              }
```

```
9019              }
9020          \cs_new_protected:cpx { msg_ #1 :nnnnn } ##1##2##3##4##5
9021            { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
9022          \cs_new_protected:cpx { msg_ #1 :nnnn } ##1##2##3##4
9023            { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
9024          \cs_new_protected:cpx { msg_ #1 :nnn } ##1##2##3
9025            { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
9026          \cs_new_protected:cpx { msg_ #1 :nn } ##1##2
9027            { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
9028          \cs_new_protected:cpx { msg_ #1 :nnxxxx } ##1##2##3##4##5##6
9029            {
9030              \use:x
9031                {
9032                  \exp_not:N \exp_not:n
9033                    { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} }
9034                    {##3} {##4} {##5} {##6}
9035                }
9036            }
9037          \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
9038            { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
9039          \cs_new_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
9040            { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
9041          \cs_new_protected:cpx { msg_ #1 :nnx } ##1##2##3
9042            { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
9043      }
```

(*End definition for* `\__msg_class_new:nn`.)

\msg_fatal:nnnnnn For fatal errors, after the error message TeX bails out.
\msg_fatal:nnxxxx
\msg_fatal:nnnnn
\msg_fatal:nnxxx
\msg_fatal:nnnn
\msg_fatal:nnxx
\msg_fatal:nnn
\msg_fatal:nnx
\msg_fatal:nn

```
9044      \__msg_class_new:nn { fatal }
9045        {
9046          \msg_interrupt:nnn
9047            { \msg_fatal_text:n {#1} : ~ "#2" }
9048            {
9049              \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
9050              \msg_see_documentation_text:n {#1}
9051            }
9052            { \c__msg_fatal_text_tl }
9053          \tex_end:D
9054        }
```

(*End definition for* `\msg_fatal:nnnnnn` *and others. These functions are documented on page 152.*)

\msg_critical:nnnnnn Not quite so bad: just end the current file.
\msg_critical:nnxxxx
\msg_critical:nnnnn
\msg_critical:nnxxx
\msg_critical:nnnn
\msg_critical:nnxx
\msg_critical:nnn
\msg_critical:nnx
\msg_critical:nn

```
9055      \__msg_class_new:nn { critical }
9056        {
9057          \msg_interrupt:nnn
9058            { \msg_critical_text:n {#1} : ~ "#2" }
9059            {
9060              \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
9061              \msg_see_documentation_text:n {#1}
9062            }
9063            { \c__msg_critical_text_tl }
9064          \tex_endinput:D
9065        }
```

483

(End definition for `\msg_critical:nnnnnn` *and others. These functions are documented on page 152.)*

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.

```
9066    \__msg_class_new:nn { error }
9067      {
9068        \__msg_error:cnnnnn
9069          { \c__msg_more_text_prefix_tl #1 / #2 }
9070          {#3} {#4} {#5} {#6}
9071          {
9072            \msg_interrupt:nnn
9073              { \msg_error_text:n {#1} : ~ "#2" }
9074              {
9075                \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
9076                \msg_see_documentation_text:n {#1}
9077              }
9078          }
9079      }
9080    \cs_new_protected:Npn \__msg_error:cnnnnn #1#2#3#4#5#6
9081      {
9082        \cs_if_eq:cNTF {#1} \__msg_no_more_text:nnnn
9083          { #6 { } }
9084          { #6 { \use:c {#1} {#2} {#3} {#4} {#5} } }
9085      }
9086    \cs_new:Npn \__msg_no_more_text:nnnn #1#2#3#4 { }
```

(End definition for `\msg_error:nnnnnn` *and others. These functions are documented on page 153.)*

Warnings are printed to the terminal.

```
9087    \__msg_class_new:nn { warning }
9088      {
9089        \msg_term:n
9090          {
9091            \msg_warning_text:n {#1} : ~ "#2" \\ \\
9092            \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
9093          }
9094      }
```

(End definition for `\msg_warning:nnnnnn` *and others. These functions are documented on page 153.)*

Information only goes into the log.

```
9095    \__msg_class_new:nn { info }
9096      {
9097        \msg_log:n
9098          {
9099            \msg_info_text:n {#1} : ~ "#2" \\ \\
9100            \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
9101          }
9102      }
```

(End definition for `\msg_info:nnnnnn` *and others. These functions are documented on page 153.)*

"Log" data is very similar to information, but with no extras added.

```
9103    \__msg_class_new:nn { log }
9104      {
9105        \iow_wrap:nnnN
9106          { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
9107          { } { } \iow_log:n
9108      }
```

(*End definition for* \msg_log:nnnnnn *and others. These functions are documented on page 153.*)

The none message type is needed so that input can be gobbled.

```
9109    \__msg_class_new:nn { none } { }
```

(*End definition for* \msg_none:nnnnnn *and others. These functions are documented on page 154.*)

End the group to eliminate __msg_class_new:nn.

```
9110  \group_end:
```

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.

```
9111  \cs_new:Npn \__msg_class_chk_exist:nT #1
9112    {
9113      \cs_if_free:cTF { __msg_ #1 _code:nnnnnn }
9114        { \__msg_kernel_error:nnx { kernel } { message-class-unknown } {#1} }
9115    }
```

(*End definition for* __msg_class_chk_exist:nT.)

\l__msg_class_tl   Support variables needed for the redirection system.
\l__msg_current_class_tl

```
9116  \tl_new:N \l__msg_class_tl
9117  \tl_new:N \l__msg_current_class_tl
```

(*End definition for* \l__msg_class_tl *and* \l__msg_current_class_tl.)

\l__msg_redirect_prop   For redirection of individually-named messages

```
9118  \prop_new:N \l__msg_redirect_prop
```

(*End definition for* \l__msg_redirect_prop.)

\l__msg_hierarchy_seq   During redirection, split the message name into a sequence with items {/module/submodule}, {/module}, and {}.

```
9119  \seq_new:N \l__msg_hierarchy_seq
```

(*End definition for* \l__msg_hierarchy_seq.)

\l__msg_class_loop_seq   Classes encountered when following redirections to check for loops.

```
9120  \seq_new:N \l__msg_class_loop_seq
```

(*End definition for* \l__msg_class_loop_seq.)

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.

```
9121 \cs_new_protected:Npn \__msg_use:nnnnnnn #1#2#3#4#5#6#7
9122   {
9123     \msg_if_exist:nnTF {#2} {#3}
9124       {
9125         \__msg_class_chk_exist:nT {#1}
9126           {
9127             \tl_set:Nn \l__msg_current_class_tl {#1}
9128             \cs_set_protected:Npx \__msg_use_code:
9129               {
9130                 \exp_not:n
9131                   {
9132                     \use:c { __msg_ \l__msg_class_tl _code:nnnnnn }
9133                       {#2} {#3} {#4} {#5} {#6} {#7}
9134                   }
9135               }
9136             \__msg_use_redirect_name:n { #2 / #3 }
9137           }
9138       }
9139       { \__msg_kernel_error:nnxx { kernel } { message-unknown } {#2} {#3} }
9140   }
9141 \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/message (with an arbitrary number of slashes), and store {/module/submodule}, {/module} and {} into \l__msg_hierarchy_seq. We will then map through this sequence, applying the most specific redirection.

```
9142 \cs_new_protected:Npn \__msg_use_redirect_name:n #1
9143   {
9144     \prop_get:NnNTF \l__msg_redirect_prop { / #1 } \l__msg_class_tl
9145       { \__msg_use_code: }
9146       {
9147         \seq_clear:N \l__msg_hierarchy_seq
9148         \__msg_use_hierarchy:nwwN { }
9149           #1 \q_mark \__msg_use_hierarchy:nwwN
9150           /  \q_mark \use_none_delimit_by_q_stop:w
9151         \q_stop
9152         \__msg_use_redirect_module:n { }
9153       }
9154   }
9155 \cs_new_protected:Npn \__msg_use_hierarchy:nwwN #1#2 / #3 \q_mark #4
9156   {
9157     \seq_put_left:Nn \l__msg_hierarchy_seq {#1}
9158     #4 { #1 / #2 } #3 \q_mark #4
9159   }
```

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

486

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`.

```
9160 \cs_new_protected:Npn \__msg_use_redirect_module:n #1
9161   {
9162     \seq_map_inline:Nn \l__msg_hierarchy_seq
9163       {
9164         \prop_get:cnNTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
9165           {##1} \l__msg_class_tl
9166           {
9167             \seq_map_break:n
9168               {
9169                 \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
9170                   { \__msg_use_code: }
9171                   {
9172                     \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
9173                     \__msg_use_redirect_module:n {##1}
9174                   }
9175               }
9176           }
9177           {
9178             \str_if_eq:nnT {##1} {#1}
9179               {
9180                 \tl_set_eq:NN \l__msg_class_tl \l__msg_current_class_tl
9181                 \seq_map_break:n { \__msg_use_code: }
9182               }
9183           }
9184       }
9185   }
```

(*End definition for* `\__msg_use:nnnnnnn` *and others.*)

`\msg_redirect_name:nnn` Named message will always use the given class even if that class is redirected further. An empty target class cancels any existing redirection for that message.

```
9186 \cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
9187   {
9188     \tl_if_empty:nTF {#3}
9189       { \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 } }
9190       {
9191         \__msg_class_chk_exist:nT {#3}
9192           { \prop_put:Nnn \l__msg_redirect_prop { / #1 / #2 } {#3} }
9193       }
9194   }
```

(*End definition for* `\msg_redirect_name:nnn`. *This function is documented on page 155.*)

`\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`.

```
9195 \cs_new_protected:Npn \msg_redirect_class:nn
9196   { \__msg_redirect:nnn { } }
9197 \cs_new_protected:Npn \msg_redirect_module:nnn #1
9198   { \__msg_redirect:nnn { / #1 } }
9199 \cs_new_protected:Npn \__msg_redirect:nnn #1#2#3
9200   {
9201     \__msg_class_chk_exist:nT {#2}
9202       {
9203         \tl_if_empty:nTF {#3}
9204           { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
9205           {
9206             \__msg_class_chk_exist:nT {#3}
9207               {
9208                 \prop_put:cnn { l__msg_redirect_ #2 _prop } {#1} {#3}
9209                 \tl_set:Nn \l__msg_current_class_tl {#2}
9210                 \seq_clear:N \l__msg_class_loop_seq
9211                 \__msg_redirect_loop_chk:nnn {#2} {#3} {#1}
9212               }
9213           }
9214       }
9215   }
```

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.

```
9216 \cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
9217   {
9218     \seq_put_right:Nn \l__msg_class_loop_seq {#1}
9219     \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_class_tl
9220       {
9221         \str_if_eq_x:nnF { \l__msg_class_tl } {#1}
9222           {
9223             \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl
9224               {
9225                 \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
9226                 \__msg_kernel_warning:nnxxxx
9227                   { kernel } { message-redirect-loop }
9228                   { \seq_item:Nn \l__msg_class_loop_seq { \c_one } }
9229                   { \seq_item:Nn \l__msg_class_loop_seq { \c_two } }
9230                   {#3}
9231                   {
9232                     \seq_map_function:NN \l__msg_class_loop_seq
9233                       \__msg_redirect_loop_list:n
9234                     { \seq_item:Nn \l__msg_class_loop_seq { \c_one } }
9235                   }
9236               }
```

```
9237                    { \__msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} }
9238                }
9239            }
9240        }
9241    \cs_generate_variant:Nn \__msg_redirect_loop_chk:nnn { o }
9242    \cs_new:Npn \__msg_redirect_loop_list:n #1 { {#1} ~ => ~ }
```

(*End definition for* `\msg_redirect_class:nn` *and others. These functions are documented on page 154.*)

18.5 Kernel-specific functions

`\__msg_kernel_new:nnnn`
`\__msg_kernel_new:nnn`
`\__msg_kernel_set:nnnn`
`\__msg_kernel_set:nnn`

The kernel needs some messages of its own. These are created using pre-built functions. Two functions are provided: one more general and one which only has the short text part.

```
9243    \cs_new_protected:Npn \__msg_kernel_new:nnnn #1#2
9244        { \msg_new:nnnn { LaTeX } { #1 / #2 } }
9245    \cs_new_protected:Npn \__msg_kernel_new:nnn #1#2
9246        { \msg_new:nnn { LaTeX } { #1 / #2 } }
9247    \cs_new_protected:Npn \__msg_kernel_set:nnnn #1#2
9248        { \msg_set:nnnn { LaTeX } { #1 / #2 } }
9249    \cs_new_protected:Npn \__msg_kernel_set:nnn #1#2
9250        { \msg_set:nnn { LaTeX } { #1 / #2 } }
```

(*End definition for* `\__msg_kernel_new:nnnn` *and others.*)

`\__msg_kernel_class_new:nN`
`\__msg_kernel_class_new_aux:nN`

All the functions for kernel messages come in variants ranging from 0 to 4 arguments. Those with less than 4 arguments are defined in terms of the 4-argument variant, in a way very similar to `\__msg_class_new:nn`. This auxiliary is destroyed at the end of the group.

```
9251    \group_begin:
9252        \cs_set_protected:Npn \__msg_kernel_class_new:nN #1
9253            { \__msg_kernel_class_new_aux:nN { kernel_ #1 } }
9254        \cs_set_protected:Npn \__msg_kernel_class_new_aux:nN #1#2
9255            {
9256                \cs_new_protected:cpn { __msg_ #1 :nnnnnn } ##1##2##3##4##5##6
9257                    {
9258                        \use:x
9259                            {
9260                                \exp_not:n { #2 { LaTeX } { ##1 / ##2 } }
9261                                    { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
9262                                    { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
9263                            }
9264                    }
9265                \cs_new_protected:cpx { __msg_ #1 :nnnnn } ##1##2##3##4##5
9266                    { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
9267                \cs_new_protected:cpx { __msg_ #1 :nnnn } ##1##2##3##4
9268                    { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
9269                \cs_new_protected:cpx { __msg_ #1 :nnn } ##1##2##3
9270                    { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
9271                \cs_new_protected:cpx { __msg_ #1 :nn } ##1##2
9272                    { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
9273                \cs_new_protected:cpx { __msg_ #1 :nnxxxx } ##1##2##3##4##5##6
9274                    {
9275                        \use:x
```

489

```
9276                    {
9277                      \exp_not:N \exp_not:n
9278                        { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} }
9279                        {##3} {##4} {##5} {##6}
9280                    }
9281                }
9282            \cs_new_protected:cpx { __msg_ #1 :nnxxx } ##1##2##3##4##5
9283              { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
9284            \cs_new_protected:cpx { __msg_ #1 :nnxx } ##1##2##3##4
9285              { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
9286            \cs_new_protected:cpx { __msg_ #1 :nnx } ##1##2##3
9287              { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
9288        }
```

(*End definition for* `\__msg_kernel_class_new:nN` *and* `\__msg_kernel_class_new_aux:nN`.)

Neither fatal kernel errors nor kernel errors can be redirected. We directly use the code for (non-kernel) fatal errors and errors, adding the "LaTeX" module name. Three functions are already defined by l3basics; we need to undefine them to avoid errors.

```
9289    \__msg_kernel_class_new:nN { fatal } \__msg_fatal_code:nnnnnn
9290    \cs_undefine:N \__msg_kernel_error:nnxx
9291    \cs_undefine:N \__msg_kernel_error:nnx
9292    \cs_undefine:N \__msg_kernel_error:nn
9293    \__msg_kernel_class_new:nN { error } \__msg_error_code:nnnnnn
```

(*End definition for* `\__msg_kernel_fatal:nnnnnn` *and others.*)

Kernel messages which can be redirected simply use the machinery for normal messages, with the module name "LaTeX".

```
9294    \__msg_kernel_class_new:nN { warning } \msg_warning:nnxxxx
9295    \__msg_kernel_class_new:nN { info } \msg_info:nnxxxx
```

(*End definition for* `\__msg_kernel_warning:nnnnnn` *and others.*)

End the group to eliminate `\__msg_kernel_class_new:nN`.

```
9296 \group_end:
```

Error messages needed to actually implement the message system itself.

```
9297 \__msg_kernel_new:nnnn { kernel } { message-already-defined }
9298    { Message~'#2'~for~module~'#1'~already~defined. }
9299    {
9300      \c__msg_coding_error_text_tl
9301      LaTeX~was~asked~to~define~a~new~message~called~'#2'\\
9302      by~the~module~'#1':~this~message~already~exists.
9303      \c__msg_return_text_tl
9304    }
9305 \__msg_kernel_new:nnnn { kernel } { message-unknown }
9306    { Unknown~message~'#2'~for~module~'#1'. }
9307    {
9308      \c__msg_coding_error_text_tl
9309      LaTeX~was~asked~to~display~a~message~called~'#2'\\
9310      by~the~module~'#1':~this~message~does~not~exist.
9311      \c__msg_return_text_tl
9312    }
9313 \__msg_kernel_new:nnnn { kernel } { message-class-unknown }
9314    { Unknown~message~class~'#1'. }
```

490

```
9315    {
9316      LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\\
9317      this~was~never~defined.
9318      \c__msg_return_text_tl
9319    }
9320  \__msg_kernel_new:nnnn { kernel } { message-redirect-loop }
9321    {
9322      Message~redirection~loop~caused~by~ {#1} ~=>~ {#2}
9323      \tl_if_empty:nF {#3} { ~for~module~' \use_none:n #3 ' } .
9324    }
9325    {
9326      Adding~the~message~redirection~ {#1} ~=>~ {#2}
9327      \tl_if_empty:nF {#3} { ~for~the~module~' \use_none:n #3 ' } ~
9328      created~an~infinite~loop\\\\
9329      \iow_indent:n { #4 \\\\ }
9330    }
```

Messages for earlier kernel modules.

```
9331  \__msg_kernel_new:nnnn { kernel } { bad-number-of-arguments }
9332    { Function~'#1'~cannot~be~defined~with~#2~arguments. }
9333    {
9334      \c__msg_coding_error_text_tl
9335      LaTeX~has~been~asked~to~define~a~function~'#1'~with~
9336      #2~arguments.~
9337      TeX~allows~between~0~and~9~arguments~for~a~single~function.
9338    }
9339  \__msg_kernel_new:nnn { kernel } { char-active }
9340    { Cannot~generate~active~chars. }
9341  \__msg_kernel_new:nnn { kernel } { char-invalid-catcode }
9342    { Invalid~catcode~for~char~generation. }
9343  \__msg_kernel_new:nnn { kernel } { char-null-space }
9344    { Cannot~generate~null~char~as~a~space. }
9345  \__msg_kernel_new:nnn { kernel } { char-out-of-range }
9346    { Charcode~requested~out~of~engine~range. }
9347  \__msg_kernel_new:nnn { kernel } { char-space }
9348    { Cannot~generate~space~chars. }
9349  \__msg_kernel_new:nnnn { kernel } { command-already-defined }
9350    { Control~sequence~#1~already~defined. }
9351    {
9352      \c__msg_coding_error_text_tl
9353      LaTeX~has~been~asked~to~create~a~new~control~sequence~'#1'~
9354      but~this~name~has~already~been~used~elsewhere. \\ \\
9355      The~current~meaning~is:\\
9356      \ \ #2
9357    }
9358  \__msg_kernel_new:nnnn { kernel } { command-not-defined }
9359    { Control~sequence~#1~undefined. }
9360    {
9361      \c__msg_coding_error_text_tl
9362      LaTeX~has~been~asked~to~use~a~control~sequence~'#1':\\
9363      this~has~not~been~defined~yet.
9364    }
9365  \__msg_kernel_new:nnnn { kernel } { empty-search-pattern }
9366    { Empty~search~pattern. }
9367    {
```

491

```
9368        \c__msg_coding_error_text_tl
9369        LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'#1':~that~
9370        would~lead~to~an~infinite~loop!
9371      }
9372 \__msg_kernel_new:nnnn { kernel } { out-of-registers }
9373      { No~room~for~a~new~#1. }
9374      {
9375        TeX~only~supports~\int_use:N \c_max_register_int \ %
9376        of~each~type.~All~the~#1~registers~have~been~used.~
9377        This~run~will~be~aborted~now.
9378      }
9379 \__msg_kernel_new:nnnn { kernel } { non-base-function }
9380      { Function~'#1'~is~not~a~base~function }
9381      {
9382        \c__msg_coding_error_text_tl
9383        Functions~defined~through~\iow_char:N\\cs_new:Nn~must~have~
9384        a~signature~consisting~of~only~normal~arguments~'N'~and~'n'.~
9385        To~define~variants~use~\iow_char:N\\cs_generate_variant:Nn~
9386        and~to~define~other~functions~use~\iow_char:N\\cs_new:Npn.
9387      }
9388 \__msg_kernel_new:nnnn { kernel } { missing-colon }
9389      { Function~'#1'~contains~no~':'. }
9390      {
9391        \c__msg_coding_error_text_tl
9392        Code-level~functions~must~contain~':'~to~separate~the~
9393        argument~specification~from~the~function~name.~This~is~
9394        needed~when~defining~conditionals~or~variants,~or~when~building~a~
9395        parameter~text~from~the~number~of~arguments~of~the~function.
9396      }
9397 \__msg_kernel_new:nnnn { kernel } { protected-predicate }
9398      { Predicate~'#1'~must~be~expandable. }
9399      {
9400        \c__msg_coding_error_text_tl
9401        LaTeX~has~been~asked~to~define~'#1'~as~a~protected~predicate.~
9402        Only~expandable~tests~can~have~a~predicate~version.
9403      }
9404 \__msg_kernel_new:nnnn { kernel } { conditional-form-unknown }
9405      { Conditional~form~'#1'~for~function~'#2'~unknown. }
9406      {
9407        \c__msg_coding_error_text_tl
9408        LaTeX~has~been~asked~to~define~the~conditional~form~'#1'~of~
9409        the~function~'#2',~but~only~'TF',~'T',~'F',~and~'p'~forms~exist.
9410      }
9411 ⟨*package⟩
9412 \bool_if:NT \l@expl@check@declarations@bool
9413      {
9414        \__msg_kernel_new:nnnn { check } { non-declared-variable }
9415          { The~variable~#1~has~not~been~declared~\msg_line_context:. }
9416          {
9417            Checking~is~active,~and~you~have~tried~do~so~something~like: \\
9418            \ \ \tl_set:Nn ~ #1 ~ \{ ~ ... ~ \} \\
9419            without~first~having: \\
9420            \ \ \tl_new:N ~ #1   \\
9421            \\
```

492

```
9422        LaTeX~will~create~the~variable~and~continue.
9423      }
9424    }
9425  ⟨/package⟩
9426  \__msg_kernel_new:nnnn { kernel } { scanmark-already-defined }
9427    { Scan~mark~#1~already~defined. }
9428    {
9429      \c__msg_coding_error_text_tl
9430      LaTeX~has~been~asked~to~create~a~new~scan~mark~'#1'~
9431      but~this~name~has~already~been~used~for~a~scan~mark.
9432    }
9433  \__msg_kernel_new:nnnn { kernel } { variable-not-defined }
9434    { Variable~#1~undefined. }
9435    {
9436      \c__msg_coding_error_text_tl
9437      LaTeX~has~been~asked~to~show~a~variable~#1,~but~this~has~not~
9438      been~defined~yet.
9439    }
9440  \__msg_kernel_new:nnnn { kernel } { variant-too-long }
9441    { Variant~form~'#1'~longer~than~base~signature~of~'#2'. }
9442    {
9443      \c__msg_coding_error_text_tl
9444      LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
9445      with~a~signature~starting~with~'#1',~but~that~is~longer~than~
9446      the~signature~(part~after~the~colon)~of~'#2'.
9447    }
9448  \__msg_kernel_new:nnnn { kernel } { invalid-variant }
9449    { Variant~form~'#1'~invalid~for~base~form~'#2'. }
9450    {
9451      \c__msg_coding_error_text_tl
9452      LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
9453      with~a~signature~starting~with~'#1',~but~cannot~change~an~argument~
9454      from~type~'#3'~to~type~'#4'.
9455    }
```

Some errors only appear in expandable settings, hence don't need a "more-text" argument.

```
9456  \__msg_kernel_new:nnn { kernel } { bad-variable }
9457    { Erroneous~variable~#1 used! }
9458  \__msg_kernel_new:nnn { kernel } { misused-sequence }
9459    { A~sequence~was~misused. }
9460  \__msg_kernel_new:nnn { kernel } { misused-prop }
9461    { A~property~list~was~misused. }
9462  \__msg_kernel_new:nnn { kernel } { negative-replication }
9463    { Negative~argument~for~\prg_replicate:nn. }
9464  \__msg_kernel_new:nnn { kernel } { unknown-comparison }
9465    { Relation~'#1'~unknown:~use~=,~<,~>,~==,~!=,~<=,~>=. }
9466  \__msg_kernel_new:nnn { kernel } { zero-step }
9467    { Zero~step~size~for~step~function~#1. }
```

Messages used by the "show" functions.

```
9468  \__msg_kernel_new:nnn { kernel } { show-clist }
9469    {
9470      The~comma~list~ \tl_if_empty:nF {#1} { #1 ~ }
9471      \tl_if_empty:nTF {#2}
```

```
9472          { is~empty }
9473          { contains~the~items~(without~outer~braces): }
9474      }
9475  \__msg_kernel_new:nnn { kernel } { show-prop }
9476      {
9477        The~property~list~#1~
9478        \tl_if_empty:nTF {#2}
9479          { is~empty }
9480          { contains~the~pairs~(without~outer~braces): }
9481      }
9482  \__msg_kernel_new:nnn { kernel } { show-seq }
9483      {
9484        The~sequence~#1~
9485        \tl_if_empty:nTF {#2}
9486          { is~empty }
9487          { contains~the~items~(without~outer~braces): }
9488      }
9489  \__msg_kernel_new:nnn { kernel } { show-streams }
9490      {
9491        \tl_if_empty:nTF {#2} { No~ } { The~following~ }
9492        \str_case:nn {#1}
9493          {
9494            { ior } { input ~ }
9495            { iow } { output ~ }
9496          }
9497        streams~are~
9498        \tl_if_empty:nTF {#2} { open } { in~use: }
9499      }
```

18.6   Expandable errors

In expansion only context, we cannot use the normal means of reporting errors. Instead, we feed TeX an undefined control sequence, `\LaTeX3 error:`. It is thus interrupted, and shows the context, which thanks to the odd-looking `\use:n` is

```
<argument> \LaTeX3 error:
                              The error message.
```

In other words, TeX is processing the argument of `\use:n`, which is `\LaTeX3 error:` ⟨*error message*⟩. Then `\__msg_expandable_error:w` cleans up. In fact, there is an extra subtlety: if the user inserts tokens for error recovery, they should be kept. Thus we also use an odd space character (with category code 7) and keep tokens until that space character, dropping everything else until `\q_stop`. The `\exp_end:` prevents losing braces around the user-inserted text if any, and stops the expansion of `\exp:w`. The group is used to prevent `\LaTeX3~error:` from being globally equal to `\scan_stop:`.

```
9500  \group_begin:
9501  \cs_set_protected:Npn \__msg_tmp:w #1#2
9502      {
9503        \cs_new:Npn \__msg_expandable_error:n ##1
9504          {
9505            \exp:w
9506            \exp_after:wN \exp_after:wN
9507            \exp_after:wN \__msg_expandable_error:w
```

```
9508          \exp_after:wN \exp_after:wN
9509          \exp_after:wN \exp_end:
9510          \use:n { #1 #2 ##1 } #2
9511        }
9512      \cs_new:Npn \__msg_expandable_error:w ##1 #2 ##2 #2 {##1}
9513    }
9514  \exp_args:Ncx \__msg_tmp:w { LaTeX3~error: }
9515    { \char_generate:nn { '\  } { 7 } }
9516  \group_end:
```

(*End definition for* `\__msg_expandable_error:n` *and* `\__msg_expandable_error:w`*.*)

`\__msg_kernel_expandable_error:nnnnnn`
`\__msg_kernel_expandable_error:nnnnn`
`\__msg_kernel_expandable_error:nnnn`
`\__msg_kernel_expandable_error:nnn`
`\__msg_kernel_expandable_error:nn`

The command built from the csname `\c_@@_text_prefix_tl LaTeX / #1 / #2` takes four arguments and builds the error text, which is fed to `\__msg_expandable_error:n`.

```
9517  \cs_new:Npn \__msg_kernel_expandable_error:nnnnnn #1#2#3#4#5#6
9518    {
9519      \exp_args:Nf \__msg_expandable_error:n
9520        {
9521          \exp_args:NNc \exp_after:wN \exp_stop_f:
9522            { \c__msg_text_prefix_tl LaTeX / #1 / #2 }
9523            {#3} {#4} {#5} {#6}
9524        }
9525    }
9526  \cs_new:Npn \__msg_kernel_expandable_error:nnnnn #1#2#3#4#5
9527    {
9528      \__msg_kernel_expandable_error:nnnnnn
9529        {#1} {#2} {#3} {#4} {#5} { }
9530    }
9531  \cs_new:Npn \__msg_kernel_expandable_error:nnnn #1#2#3#4
9532    {
9533      \__msg_kernel_expandable_error:nnnnnn
9534        {#1} {#2} {#3} {#4} { } { }
9535    }
9536  \cs_new:Npn \__msg_kernel_expandable_error:nnn #1#2#3
9537    {
9538      \__msg_kernel_expandable_error:nnnnnn
9539        {#1} {#2} {#3} { } { } { }
9540    }
9541  \cs_new:Npn \__msg_kernel_expandable_error:nn #1#2
9542    {
9543      \__msg_kernel_expandable_error:nnnnnn
9544        {#1} {#2} { } { } { } { }
9545    }
```

(*End definition for* `\__msg_kernel_expandable_error:nnnnnn` *and others.*)

18.7 Showing variables

Functions defined in this section are used for diagnostic functions in l3clist, l3file, l3prop, l3seq, xtemplate

`\g__msg_log_next_bool`
`\__msg_log_next:`

```
9546  \bool_new:N \g__msg_log_next_bool
9547  \cs_new_protected:Npn \__msg_log_next:
9548    { \bool_gset_true:N \g__msg_log_next_bool }
```

495

(*End definition for* \g__msg_log_next_bool *and* __msg_log_next:.)

__msg_show_pre:nnnnnn
__msg_show_pre:nnxxxx
__msg_show_pre:nnnnnV
__msg_show_pre_aux:n

Print the text of a message to the terminal or log file without formatting: short cuts around \iow_wrap:nnnN. The choice of terminal or log file is done by __msg_show_-pre_aux:n.

```
9549 \cs_new_protected:Npn \__msg_show_pre:nnnnnn #1#2#3#4#5#6
9550   {
9551     \exp_args:Nx \iow_wrap:nnnN
9552       {
9553         \exp_not:c { \c__msg_text_prefix_tl #1 / #2 }
9554           { \tl_to_str:n {#3} }
9555           { \tl_to_str:n {#4} }
9556           { \tl_to_str:n {#5} }
9557           { \tl_to_str:n {#6} }
9558       }
9559       { } { } \__msg_show_pre_aux:n
9560   }
9561 \cs_new_protected:Npn \__msg_show_pre:nnxxxx #1#2#3#4#5#6
9562   {
9563     \use:x
9564       { \exp_not:n { \__msg_show_pre:nnnnnn {#1} {#2} } {#3} {#4} {#5} {#6} }
9565   }
9566 \cs_generate_variant:Nn \__msg_show_pre:nnnnnn { nnnnnV }
9567 \cs_new_protected:Npn \__msg_show_pre_aux:n
9568   { \bool_if:NTF \g__msg_log_next_bool { \iow_log:n } { \iow_term:n } }
```

(*End definition for* __msg_show_pre:nnnnnn *and* __msg_show_pre_aux:n.)

__msg_show_variable:NNNnn The arguments of __msg_show_variable:NNNnn are

- The ⟨*variable*⟩ to be shown as #1.

- An ⟨*if-exist*⟩ conditional #2 with NTF signature.

- An ⟨*if-empty*⟩ conditional #3 or other function with NTF signature (sometimes \use_ii:nnn).

- The ⟨*message*⟩ #4 to use.

- A construction #5 which produces the formatted string eventually passed to the \showtokens primitive. Typically this is a mapping of the form \seq_map_-function:NN ⟨*variable*⟩ __msg_show_item:n.

If ⟨*if-exist*⟩ ⟨*variable*⟩ is false, throw an error and remember to reset \g__msg_log_-next_bool, which is otherwise reset by __msg_show_wrap:n. If ⟨*message*⟩ is not empty, output the message LaTeX/kernel/show-⟨*message*⟩ with as its arguments the ⟨*variable*⟩, and either an empty second argument or ? depending on the result of ⟨*if-empty*⟩ ⟨*variable*⟩. Afterwards, show the contents of #5 using __msg_show_wrap:n or __-msg_log_wrap:n.

```
9569 \cs_new_protected:Npn \__msg_show_variable:NNNnn #1#2#3#4#5
9570   {
9571     #2 #1
9572       {
9573         \tl_if_empty:nF {#4}
9574           {
```

496

```
9575            \__msg_show_pre:nnxxxx { LaTeX / kernel } { show- #4 }
9576              { \token_to_str:N #1 } { #3 #1 { } { ? } } { } { }
9577          }
9578        \__msg_show_wrap:n {#5}
9579      }
9580      {
9581        \__msg_kernel_error:nnx { kernel } { variable-not-defined }
9582          { \token_to_str:N #1 }
9583        \bool_gset_false:N \g__msg_log_next_bool
9584      }
9585    }
```

(*End definition for* `\__msg_show_variable:NNNnn`.)

__msg_show_wrap:Nn A short-hand used for `\int_show:n` and many other functions that passes to `\__msg_-`
`show_wrap:n` the result of applying `#1` (a function such as `\int_eval:n`) to the expression
`#2`. The leading `>~` is needed by `\__msg_show_wrap:n`. The use of x-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`. This does not lead to double expansion
because the x-expansion of `#1 {#2}` is a string in all cases where `\__msg_show_wrap:Nn`
is used.

```
9586 \cs_new_protected:Npn \__msg_show_wrap:Nn #1#2
9587   { \exp_args:Nx \__msg_show_wrap:n { > ~ \tl_to_str:n {#2} = #1 {#2} } }
```

(*End definition for* `\__msg_show_wrap:Nn`.)

__msg_show_wrap:n The argument of `\__msg_show_wrap:n` is line-wrapped using `\iow_wrap:nnnN`. Every-
__msg_show_wrap_aux:n thing before the first `>` in the wrapped text is removed, as well as an optional space
__msg_show_wrap_aux:w following it (because of f-expansion). In order for line-wrapping to give the correct re-
sult, the first `>` must in fact appear at the beginning of a line and be followed by a space
(or a line-break), so in practice, the argument of `\__msg_show_wrap:n` begins with `>~` or
`\\>~`.

The line-wrapped text is then either sent to the log file through `\iow_log:x`, or
shown in the terminal using the ε-TeX primitive `\showtokens` after removing a leading
`>~` and trailing dot since those are added automatically by `\showtokens`. The trailing dot
was included in the first place because its presence can affect line-wrapping. Note that the
space after `>` is removed through f-expansion rather than by using an argument delimited
by `>~` because the space may have been replaced by a line-break when line-wrapping.

A special case is that if the line-wrapped text is a single dot (in other words if the
argument of `\__msg_show_wrap:n` x-expands to nothing) then no `>~` should be removed.
This makes it unnecessary to check explicitly for emptyness when using for instance
`\seq_map_function:NN` ⟨*seq var*⟩ `\__msg_show_item:n` as the argument of `\__msg_-`
`show_wrap:n`.

Finally, the token list `\l__msg_internal_tl` containing the result of all these ma-
nipulations is displayed to the terminal using `\etex_showtokens:D` and odd `\exp_-`
`after:wN` which expand the closing brace to improve the output slightly. The calls
to `\__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.

Note also that `\g__msg_log_next_bool` is only reset if that is necessary. This allows
the user of an interactive prompt to insert tokens as a response to ε-TeX's `\showtokens`.

```
9588 \cs_new_protected:Npn \__msg_show_wrap:n #1
9589   { \iow_wrap:nnnN { #1 . } { } { } \__msg_show_wrap_aux:n }
9590 \cs_new_protected:Npn \__msg_show_wrap_aux:n #1
9591   {
9592     \tl_if_single:nTF {#1}
9593       { \tl_clear:N \l__msg_internal_tl }
9594       { \tl_set:Nf \l__msg_internal_tl { \__msg_show_wrap_aux:w #1 \q_stop } }
9595     \bool_if:NTF \g__msg_log_next_bool
9596       {
9597         \iow_log:x { > ~ \l__msg_internal_tl . }
9598         \bool_gset_false:N \g__msg_log_next_bool
9599       }
9600       {
9601         \__iow_with:Nnn \tex_newlinechar:D { 10 }
9602           {
9603             \__iow_with:Nnn \tex_errorcontextlines:D \c_minus_one
9604               {
9605                 \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
9606                   { \exp_after:wN \l__msg_internal_tl }
9607               }
9608           }
9609       }
9610   }
9611 \cs_new:Npn \__msg_show_wrap_aux:w #1 > #2 . \q_stop {#2}
```

(*End definition for* __msg_show_wrap:n *,* __msg_show_wrap_aux:n *, and* __msg_show_wrap_aux:w *.*)

__msg_show_item:n
__msg_show_item:nn
__msg_show_item_unbraced:nn

Each item in the variable is formatted using one of the following functions.

```
9612 \cs_new:Npn \__msg_show_item:n #1
9613   {
9614     \\ > \ \ \{ \tl_to_str:n {#1} \}
9615   }
9616 \cs_new:Npn \__msg_show_item:nn #1#2
9617   {
9618     \\ > \ \ \{ \tl_to_str:n {#1} \}
9619     \ \ => \ \ \{ \tl_to_str:n {#2} \}
9620   }
9621 \cs_new:Npn \__msg_show_item_unbraced:nn #1#2
9622   {
9623     \\ > \ \ \tl_to_str:n {#1}
9624     \ \ => \ \ \tl_to_str:n {#2}
9625   }
```

(*End definition for* __msg_show_item:n *,* __msg_show_item:nn *, and* __msg_show_item_unbraced:nn *.*)

```
9626 ⟨/initex | package⟩
```

19 l3keys Implementation

```
9627 ⟨*initex | package⟩
```

19.1 Low-level interface

The low-level key parser is based heavily on keyval, but with a number of additional
"safety" requirements and with the idea that the parsed list of key–value pairs can be

processed in a variety of ways. The net result is that this code needs around twice 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.

```
9628 ⟨@@=keyval⟩
```

\l__keyval_key_tl
\l__keyval_value_tl

The current key name and value.

```
9629 \tl_new:N \l__keyval_key_tl
9630 \tl_new:N \l__keyval_value_tl
```

(*End definition for* \l__keyval_key_tl *and* \l__keyval_value_tl.)

\l__keyval_sanitise_tl   A token list variable for dealing with awkward category codes in the input.

```
9631 \tl_new:N \l__keyval_sanitise_tl
```

(*End definition for* \l__keyval_sanitise_tl.)

\keyval_parse:NNn   The main function starts off by normalising category codes in package mode. That's relatively "expensive" so is skipped (hopefully) in format mode. We then hand off to the parser. The use of \q_mark here prevents loss of braces from the key argument. This particular quark is chosen as it fits in with __tl_trim_spaces:nn and allows a performance enhancement as the token can be carried through. Notice that by passing the two processor commands along the input stack we avoid the need to track these at all.

```
9632 \cs_new_protected:Npn \keyval_parse:NNn #1#2#3
9633   {
9634 ⟨*initex⟩
9635     \__keyval_loop:NNw #1#2 \q_mark #3 , \q_recursion_tail ,
9636 ⟨/initex⟩
9637 ⟨*package⟩
9638     \tl_set:Nn \l__keyval_sanitise_tl {#3}
9639     \__keyval_sanitise_equals:
9640     \__keyval_sanitise_comma:
9641     \exp_after:wN \__keyval_loop:NNw \exp_after:wN #1 \exp_after:wN #2
9642       \exp_after:wN \q_mark \l__keyval_sanitise_tl , \q_recursion_tail ,
9643 ⟨/package⟩
9644   }
```

(*End definition for* \keyval_parse:NNn. *This function is documented on page 171.*)

__keyval_sanitise_equals:
__keyval_sanitise_comma:
__keyval_sanitise_equals_auxi:w
__keyval_sanitise_equals_auxii:w
__keyval_sanitise_comma_auxi:w
__keyval_sanitise_comma_auxii:w
__keyval_sanitise_aux:w

A reasonably fast search and replace set up specifically for the active tokens. The nature of the input is known so everything is hard-coded. With only two tokens to cover, the speed gain from using dedicated functions is worth it.

```
9645 ⟨*package⟩
9646 \group_begin:
9647   \char_set_catcode_active:n { '\= }
9648   \char_set_catcode_active:n { '\, }
9649   \cs_new_protected:Npn \__keyval_sanitise_equals:
9650     {
9651       \exp_after:wN \__keyval_sanitise_equals_auxi:w \l__keyval_sanitise_tl
9652         \q_mark = \q_nil =
9653       \exp_after:wN \__keyval_sanitise_aux:w \l__keyval_sanitise_tl
9654     }
```

499

```
9655     \cs_new_protected:Npn \__keyval_sanitise_equals_auxi:w #1 =
9656       {
9657         \tl_set:Nn \l__keyval_sanitise_tl {#1}
9658         \__keyval_sanitise_equals_auxii:w
9659       }
9660     \cs_new_protected:Npn \__keyval_sanitise_equals_auxii:w #1 =
9661       {
9662         \if_meaning:w \q_nil #1 \scan_stop:
9663         \else:
9664           \tl_set:Nx \l__keyval_sanitise_tl
9665             {
9666               \exp_not:o \l__keyval_sanitise_tl
9667               \token_to_str:N =
9668               \exp_not:n {#1}
9669             }
9670           \exp_after:wN \__keyval_sanitise_equals_auxii:w
9671         \fi:
9672       }
9673   \cs_new_protected:Npn \__keyval_sanitise_comma:
9674     {
9675       \exp_after:wN \__keyval_sanitise_comma_auxi:w \l__keyval_sanitise_tl
9676         \q_mark , \q_nil ,
9677       \exp_after:wN \__keyval_sanitise_aux:w \l__keyval_sanitise_tl
9678     }
9679     \cs_new_protected:Npn \__keyval_sanitise_comma_auxi:w #1 ,
9680       {
9681         \tl_set:Nn \l__keyval_sanitise_tl {#1}
9682         \__keyval_sanitise_comma_auxii:w
9683       }
9684     \cs_new_protected:Npn \__keyval_sanitise_comma_auxii:w #1 ,
9685       {
9686         \if_meaning:w \q_nil #1 \scan_stop:
9687         \else:
9688           \tl_set:Nx \l__keyval_sanitise_tl
9689             {
9690               \exp_not:o \l__keyval_sanitise_tl
9691               \token_to_str:N ,
9692               \exp_not:n {#1}
9693             }
9694           \exp_after:wN \__keyval_sanitise_comma_auxii:w
9695         \fi:
9696       }
9697 \group_end:
9698 \cs_new_protected:Npn \__keyval_sanitise_aux:w #1 \q_mark
9699   { \tl_set:Nn \l__keyval_sanitise_tl {#1} }
9700 ⟨/package⟩
```

(*End definition for* `\__keyval_sanitise_equals:` *and others.*)

`\__keyval_loop:NNw` A fast test for the end of the loop, remembering to remove the leading quark first. Assuming that is not the case, look for a key and value then loop around, re-inserting a leading quark in front of the next position.

```
9701 \cs_new_protected:Npn \__keyval_loop:NNw #1#2#3 ,
9702   {
```

```
9703        \exp_after:wN \if_meaning:w \exp_after:wN \q_recursion_tail
9704          \use_none:n #3 \prg_do_nothing:
9705        \else:
9706          \__keyval_split:NNw #1#2#3 == \q_stop
9707          \exp_after:wN \__keyval_loop:NNw \exp_after:wN #1 \exp_after:wN #2
9708            \exp_after:wN \q_mark
9709        \fi:
9710      }
```

(*End definition for* `\__keyval_loop:NNw`.)

`\__keyval_split:NNw`
`\__keyval_split_value:NNw`
`\__keyval_split_tidy:w`
`\__keyval_action:`
The value is picked up separately from the key so there can be another quark inserted at the front, keeping braces and allowing both parts to share the same code paths. The key is found first then there's a check that there is something there: this is biased to the common case of there actually being a key. For the value, we first need to see if there is anything to do: if there is, extract it. The appropriate action is then inserted in front of the key and value. Doing this using an assignment is marginally faster than an an expansion chain.

```
9711 \cs_new_protected:Npn \__keyval_split:NNw #1#2#3 =
9712   {
9713     \__keyval_def:Nn \l__keyval_key_tl {#3}
9714     \if_meaning:w \l__keyval_key_tl \c_empty_tl
9715       \exp_after:wN \__keyval_split_tidy:w
9716     \else:
9717       \exp_after:wN \__keyval_split_value:NNw \exp_after:wN #1 \exp_after:wN #2
9718         \exp_after:wN \q_mark
9719     \fi:
9720   }
9721 \cs_new_protected:Npn \__keyval_split_value:NNw #1#2#3 = #4 \q_stop
9722   {
9723     \if:w \scan_stop: \tl_to_str:n {#4} \scan_stop:
9724       \cs_set:Npx \__keyval_action:
9725         { \exp_not:N #1 { \exp_not:o \l__keyval_key_tl } }
9726     \else:
9727       \if:w \scan_stop: \etex_detokenize:D \exp_after:wN { \use_none:n #4 }
9728         \scan_stop:
9729         \__keyval_def:Nn \l__keyval_value_tl {#3}
9730         \cs_set:Npx \__keyval_action:
9731           {
9732             \exp_not:N #2
9733               { \exp_not:o \l__keyval_key_tl }
9734               { \exp_not:o \l__keyval_value_tl }
9735           }
9736       \else:
9737         \cs_set:Npn \__keyval_action:
9738           { \__msg_kernel_error:nn { kernel } { misplaced-equals-sign } }
9739       \fi:
9740     \fi:
9741     \__keyval_action:
9742   }
9743 \cs_new_protected:Npn \__keyval_split_tidy:w #1 \q_stop
9744   {
9745     \if:w \scan_stop: \etex_detokenize:D \exp_after:wN { \use_none:n #1 }
9746       \scan_stop:
```

```
9747      \else:
9748        \exp_after:wN \__keyval_empty_key:
9749      \fi:
9750    }
9751  \cs_new:Npn \__keyval_action: { }
9752  \cs_new_protected:Npn \__keyval_empty_key:
9753    { \__msg_kernel_error:nn { kernel } { misplaced-equals-sign } }
```

(*End definition for* `\__keyval_split:NNw` *and others.*)

__keyval_def:Nn
__keyval_def_aux:n
__keyval_def_aux:w

First trim spaces off, then potentially remove a set of braces. By using the internal interface `\__tl_trim_spaces:nn` we can take advantage of the fact it needs a leading `\q_mark` in this process. The `\exp_after:wN` removes the quark, the delimited argument deals with any braces.

```
9754  \cs_new_protected:Npn \__keyval_def:Nn #1#2
9755    { \tl_set:Nx #1 { \__tl_trim_spaces:nn {#2} \__keyval_def_aux:n } }
9756  \cs_new:Npn \__keyval_def_aux:n #1
9757    { \exp_after:wN \__keyval_def_aux:w #1 \q_stop }
9758  \cs_new:Npn \__keyval_def_aux:w #1 \q_stop { \exp_not:n {#1} }
```

(*End definition for* `\__keyval_def:Nn`, `\__keyval_def_aux:n`, *and* `\__keyval_def_aux:w`.)

One message for the low level parsing system.

```
9759  \__msg_kernel_new:nnnn { kernel } { misplaced-equals-sign }
9760    { Misplaced~equals~sign~in~key-value~input~\msg_line_number: }
9761    {
9762      LaTeX~is~attempting~to~parse~some~key-value~input~but~found~
9763      two~equals~signs~not~separated~by~a~comma.
9764    }
```

19.2 Constants and variables

```
9765  ⟨@@=keys⟩
```

\c__keys_code_root_tl
\c__keys_default_root_tl
\c__keys_groups_root_tl
\c__keys_info_root_tl
\c__keys_validate_root_tl

Various storage areas for the different data which make up keys.

```
9766  \tl_const:Nn \c__keys_code_root_tl      { key~code~>~ }
9767  \tl_const:Nn \c__keys_default_root_tl  { key~default~>~ }
9768  \tl_const:Nn \c__keys_groups_root_tl   { key~groups~>~ }
9769  \tl_const:Nn \c__keys_info_root_tl      { key~info~>~ }
9770  \tl_const:Nn \c__keys_validate_root_tl { key~validate~>~ }
```

(*End definition for* `\c__keys_code_root_tl` *and others.*)

\c__keys_props_root_tl

The prefix for storing properties.

```
9771  \tl_const:Nn \c__keys_props_root_tl { key~prop~>~ }
```

(*End definition for* `\c__keys_props_root_tl`.)

\l_keys_choice_int
\l_keys_choice_tl

Publicly accessible data on which choice is being used when several are generated as a set.

```
9772  \int_new:N \l_keys_choice_int
9773  \tl_new:N \l_keys_choice_tl
```

(*End definition for* `\l_keys_choice_int` *and* `\l_keys_choice_tl`. *These variables are documented on page 166.*)

\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.

9774 `\clist_new:N \l__keys_groups_clist`

(*End definition for* `\l__keys_groups_clist`.)

\l_keys_key_tl   The name of a key itself: needed when setting keys.

9775 `\tl_new:N \l_keys_key_tl`

(*End definition for* `\l_keys_key_tl`. *This variable is documented on page* *168*.)

\l__keys_module_tl   The module for an entire set of keys.

9776 `\tl_new:N \l__keys_module_tl`

(*End definition for* `\l__keys_module_tl`.)

\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.

9777 `\bool_new:N \l__keys_no_value_bool`

(*End definition for* `\l__keys_no_value_bool`.)

\l__keys_only_known_bool   Used to track if only "known" keys are being set.

9778 `\bool_new:N \l__keys_only_known_bool`

(*End definition for* `\l__keys_only_known_bool`.)

\l_keys_path_tl   The "path" of the current key is stored here: this is available to the programmer and so is public.

9779 `\tl_new:N \l_keys_path_tl`

(*End definition for* `\l_keys_path_tl`. *This variable is documented on page* *168*.)

\l__keys_property_tl   The "property" begin set for a key at definition time is stored here.

9780 `\tl_new:N \l__keys_property_tl`

(*End definition for* `\l__keys_property_tl`.)

\l__keys_selective_bool
\l_keys_filtered_bool   Two flags for using key groups: one to indicate that "selective" setting is active, a second to specify which type ("opt-in" or "opt-out").

9781 `\bool_new:N \l__keys_selective_bool`
9782 `\bool_new:N \l__keys_filtered_bool`

(*End definition for* `\l__keys_selective_bool` *and* `\l__keys_filtered_bool`.)

\l__keys_selective_seq   The list of key groups being filtered in or out during selective setting.

9783 `\seq_new:N \l__keys_selective_seq`

(*End definition for* `\l__keys_selective_seq`.)

\l__keys_unused_clist   Used when setting only some keys to store those left over.

9784 `\tl_new:N \l__keys_unused_clist`

(*End definition for* `\l__keys_unused_clist`.)

\l_keys_value_tl The value given for a key: may be empty if no value was given.

```
9785 \tl_new:N \l_keys_value_tl
```

(*End definition for \l_keys_value_tl. This variable is documented on page 168.*)

\l__keys_tmp_bool Scratch space.

```
9786 \bool_new:N \l__keys_tmp_bool
```

(*End definition for \l__keys_tmp_bool.*)

19.3 The key defining mechanism

\keys_define:nn
__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).

```
9787 \cs_new_protected:Npn \keys_define:nn
9788   { \__keys_define:onn \l_keys_module_tl }
9789 \cs_new_protected:Npn \__keys_define:nnn #1#2#3
9790   {
9791     \tl_set:Nx \l_keys_module_tl { \__keys_remove_spaces:n {#2} }
9792     \keyval_parse:NNn \__keys_define:n \__keys_define:nn {#3}
9793     \tl_set:Nn \l_keys_module_tl {#1}
9794   }
9795 \cs_generate_variant:Nn \__keys_define:nnn { o }
```

(*End definition for \keys_define:nn and __keys_define:nnn. These functions are documented on page 161.*)

__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.

```
9796 \cs_new_protected:Npn \__keys_define:n #1
9797   {
9798     \bool_set_true:N \l_keys_no_value_bool
9799     \__keys_define_aux:nn {#1} { }
9800   }
9801 \cs_new_protected:Npn \__keys_define:nn #1#2
9802   {
9803     \bool_set_false:N \l_keys_no_value_bool
9804     \__keys_define_aux:nn {#1} {#2}
9805   }
9806 \cs_new_protected:Npn \__keys_define_aux:nn #1#2
9807   {
9808     \__keys_property_find:n {#1}
9809     \cs_if_exist:cTF { \c__keys_props_root_tl \l__keys_property_tl }
9810       { \__keys_define_code:n {#2} }
9811       {
9812         \tl_if_empty:NF \l__keys_property_tl
9813           {
9814             \__msg_kernel_error:nnxx { kernel } { property-unknown }
9815               { \l__keys_property_tl } { \l_keys_path_tl }
9816           }
9817       }
9818   }
```

504

(End definition for `\__keys_define:n`*,* `\__keys_define:nn`*, and* `\__keys_define_aux:nn`*.)*

`\__keys_property_find:n`
`\__keys_property_find:w`

Searching for a property means finding the last . in the input, and storing the text before and after it. Everything is turned into strings, so there is no problem using an x-type expansion.

```
9819 \cs_new_protected:Npn \__keys_property_find:n #1
9820   {
9821     \tl_set:Nx \l__keys_property_tl { \__keys_remove_spaces:n {#1} }
9822     \exp_after:wN \__keys_property_find:w \l__keys_property_tl . . \q_stop {#1}
9823   }
9824 \cs_new_protected:Npn \__keys_property_find:w #1 . #2 . #3 \q_stop #4
9825   {
9826     \tl_if_blank:nTF {#3}
9827       {
9828         \tl_clear:N \l__keys_property_tl
9829         \__msg_kernel_error:nnn { kernel } { key-no-property } {#4}
9830       }
9831       {
9832         \str_if_eq:nnTF {#3} { . }
9833           {
9834             \tl_set:Nx \l_keys_path_tl { \l__keys_module_tl / #1 }
9835             \tl_set:Nn \l__keys_property_tl { . #2 }
9836           }
9837           {
9838             \tl_set:Nx \l_keys_path_tl { \l__keys_module_tl / #1 . #2 }
9839             \__keys_property_search:w #3 \q_stop
9840           }
9841       }
9842   }
9843 \cs_new_protected:Npn \__keys_property_search:w #1 . #2 \q_stop
9844   {
9845     \str_if_eq:nnTF {#2} { . }
9846       {
9847         \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl }
9848         \tl_set:Nn \l__keys_property_tl { . #1 }
9849       }
9850       {
9851         \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl . #1 }
9852         \__keys_property_search:w #2 \q_stop
9853       }
9854   }
```

(End definition for `\__keys_property_find:n` *and* `\__keys_property_find:w`*.)*

`\__keys_define_code:n`
`\__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. There is no need to check for a : as if it is missing the earlier tests will have failed.

```
9855 \cs_new_protected:Npn \__keys_define_code:n #1
9856   {
9857     \bool_if:NTF \l__keys_no_value_bool
9858       {
9859         \exp_after:wN \__keys_define_code:w
9860           \l__keys_property_tl \q_stop
```

505

```
9861                { \use:c { \c__keys_props_root_tl \l__keys_property_tl } }
9862                {
9863                  \__msg_kernel_error:nnxx { kernel }
9864                    { property-requires-value } { \l__keys_property_tl }
9865                    { \l_keys_path_tl }
9866                }
9867            }
9868            { \use:c { \c__keys_props_root_tl \l__keys_property_tl } {#1} }
9869      }
9870    \use:x
9871      {
9872        \cs_new:Npn \exp_not:N \__keys_define_code:w
9873          ##1 \c_colon_str ##2 \exp_not:N \q_stop
9874      }
9875      { \tl_if_empty:nTF {#2} }
```

(*End definition for* `\__keys_define_code:n` *and* `\__keys_define_code:w`.)

19.4 Turning properties into actions

`\__keys_ensure_exist:n`
`\__keys_ensure_exist:V`
Used to make sure that a key implementation and the related property list will exist whenever this is required. We cannot use for example `\prop_clear_new:c` here as that would affect the order in which key properties must be set. As key definitions are never global we use `\cs_set_protected:cpn` not `\cs_new_protected:cpn` here. For the same reason, to avoid issues if the key has been undefined in the current scope but exists at a higher level, we do not use `\prop_new:c` but rather `\prop_set_eq:cN`. The function `\__chk_log:x` only writes to the log file if logging all new functions is active: without it keys would not show up (as we are not using `\...._new`).

```
9876  \cs_new_protected:Npn \__keys_ensure_exist:n #1
9877    {
9878      \prop_if_exist:cF { \c__keys_info_root_tl #1 }
9879        {
9880          \prop_set_eq:cN { \c__keys_info_root_tl #1 } \c_empty_prop
9881        }
9882
9883      \cs_if_exist:cF { \c__keys_code_root_tl #1 }
9884        {
9885          \__chk_log:x { Defining~key~#1~ \msg_line_context: }
9886          \cs_set_protected:cpn { \c__keys_code_root_tl #1 } ##1 { }
9887        }
9888    }
9889  \cs_generate_variant:Nn \__keys_ensure_exist:n { V }
```

(*End definition for* `\__keys_ensure_exist:n`.)

`\__keys_bool_set:Nn`
`\__keys_bool_set:cn`
Boolean keys are really just choices, but all done by hand. The second argument here is the scope: either empty or `g` for global.

```
9890  \cs_new_protected:Npn \__keys_bool_set:Nn #1#2
9891    {
9892      \bool_if_exist:NF #1 { \bool_new:N #1 }
9893      \__keys_choice_make:
9894      \__keys_cmd_set:nx { \l_keys_path_tl / true }
9895        { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
```

506

```
9896        \__keys_cmd_set:nx { \l_keys_path_tl / false }
9897          { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
9898        \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
9899          {
9900            \__msg_kernel_error:nnx { kernel } { boolean-values-only }
9901              { \l_keys_key_tl }
9902          }
9903        \__keys_default_set:n { true }
9904      }
9905    \cs_generate_variant:Nn \__keys_bool_set:Nn { c }
```

(*End definition for* __keys_bool_set:Nn.)

__keys_bool_set_inverse:Nn Inverse boolean setting is much the same.
__keys_bool_set_inverse:cn

```
9906    \cs_new_protected:Npn \__keys_bool_set_inverse:Nn #1#2
9907      {
9908        \bool_if_exist:NF #1 { \bool_new:N #1 }
9909        \__keys_choice_make:
9910        \__keys_cmd_set:nx { \l_keys_path_tl / true }
9911          { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
9912        \__keys_cmd_set:nx { \l_keys_path_tl / false }
9913          { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
9914        \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
9915          {
9916            \__msg_kernel_error:nnx { kernel } { boolean-values-only }
9917              { \l_keys_key_tl }
9918          }
9919        \__keys_default_set:n { true }
9920      }
9921    \cs_generate_variant:Nn \__keys_bool_set_inverse:Nn { c }
```

(*End definition for* __keys_bool_set_inverse:Nn.)

__keys_choice_make: To make a choice from a key, two steps: set the code, and set the unknown key. There is
__keys_multichoice_make: one point to watch here: choice keys cannot be nested! As multichoices and choices are
__keys_choice_make:N essentially the same bar one function, the code is given together.
__keys_choice_make_aux:N
__keys_parent:n
__keys_parent:o
__keys_parent:wn

```
9922    \cs_new_protected:Npn \__keys_choice_make:
9923      { \__keys_choice_make:N \__keys_choice_find:n }
9924    \cs_new_protected:Npn \__keys_multichoice_make:
9925      { \__keys_choice_make:N \__keys_multichoice_find:n }
9926    \cs_new_protected:Npn \__keys_choice_make:N #1
9927      {
9928        \prop_if_exist:cTF
9929          { \c__keys_info_root_tl \__keys_parent:o \l_keys_path_tl }
9930          {
9931            \prop_get:cnNTF
9932              { \c__keys_info_root_tl \__keys_parent:o \l_keys_path_tl }
9933              { choice } \l_keys_value_tl
9934              {
9935                \__msg_kernel_error:nnxx { kernel } { nested-choice-key }
9936                  { \l_keys_path_tl } { \__keys_parent:o \l_keys_path_tl }
9937              }
9938              { \__keys_choice_make_aux:N #1 }
9939          }
```

```
9940            { \__keys_choice_make_aux:N #1 }
9941      }
9942    \cs_new_protected:Npn \__keys_choice_make_aux:N #1
9943      {
9944        \__keys_cmd_set:nn { \l_keys_path_tl } { #1 {##1} }
9945        \prop_put:cnn { \c__keys_info_root_tl \l_keys_path_tl } { choice }
9946          { true }
9947        \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
9948          {
9949            \__msg_kernel_error:nnxx { kernel } { key-choice-unknown }
9950              { \l_keys_path_tl } {##1}
9951          }
9952      }
9953    \cs_new:Npn \__keys_parent:n #1
9954      { \__keys_parent:wn #1 / / \q_stop { } }
9955    \cs_generate_variant:Nn \__keys_parent:n { o }
9956    \cs_new:Npn \__keys_parent:wn #1 / #2 / #3 \q_stop #4
9957      {
9958        \tl_if_blank:nTF {#2}
9959          { \use_none:n #4 }
9960          {
9961            \__keys_parent:wn #2 / #3 \q_stop { #4 / #1 }
9962          }
9963      }
```

(*End 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.

```
9964    \cs_new_protected:Npn \__keys_choices_make:nn
9965      { \__keys_choices_make:Nnn \__keys_choice_make: }
9966    \cs_new_protected:Npn \__keys_multichoices_make:nn
9967      { \__keys_choices_make:Nnn \__keys_multichoice_make: }
9968    \cs_new_protected:Npn \__keys_choices_make:Nnn #1#2#3
9969      {
9970        #1
9971        \int_zero:N \l_keys_choice_int
9972        \clist_map_inline:nn {#2}
9973          {
9974            \int_incr:N \l_keys_choice_int
9975            \__keys_cmd_set:nx { \l_keys_path_tl / \__keys_remove_spaces:n {##1} }
9976              {
9977                \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
9978                \int_set:Nn \exp_not:N \l_keys_choice_int
9979                  { \int_use:N \l_keys_choice_int }
9980                \exp_not:n {#3}
9981              }
9982          }
9983      }
```

(*End definition for* `\__keys_choices_make:nn`, `\__keys_multichoices_make:nn`, *and* `\__keys_choices_-`
`make:Nnn`.)

`\__keys_cmd_set:nn`
`\__keys_cmd_set:nx`
`\__keys_cmd_set:Vn`
`\__keys_cmd_set:Vo`

Setting the code for a key first checks that the basic data structures exist, then saves the code.

```
9984 \cs_new_protected:Npn \__keys_cmd_set:nn #1#2
9985   {
9986     \__keys_ensure_exist:V \l_keys_path_tl
9987     \cs_set_protected:cpn { \c__keys_code_root_tl #1 } ##1 {#2}
9988   }
9989 \cs_generate_variant:Nn \__keys_cmd_set:nn { nx , Vn , Vo }
```

(*End definition for* `\__keys_cmd_set:nn.`)

`\__keys_default_set:n` Setting a default value is easy. These are stored using `\cs_set:cpx` as this avoids any worries about whether a token list exists.

```
9990 \cs_new_protected:Npn \__keys_default_set:n #1
9991   {
9992     \tl_if_empty:nTF {#1}
9993       {
9994         \cs_set_eq:cN
9995           { \c__keys_default_root_tl \l_keys_path_tl }
9996           \tex_undefined:D
9997       }
9998       {
9999         \cs_set:cpx
10000          { \c__keys_default_root_tl \l_keys_path_tl }
10001          { \exp_not:n {#1} }
10002       }
10003   }
```

(*End definition for* `\__keys_default_set:n.`)

`\__keys_groups_set:n` Assigning a key to one or more groups uses comma lists. So that the comma list is "well-behaved" later, the storage is done via a stored list here, which does the normalisation.

```
10004 \cs_new_protected:Npn \__keys_groups_set:n #1
10005   {
10006     \__keys_ensure_exist:V \l_keys_path_tl
10007     \clist_set:Nn \l__keys_groups_clist {#1}
10008     \clist_if_empty:NTF \l__keys_groups_clist
10009       {
10010         \prop_remove:cn { \c__keys_info_root_tl \l_keys_path_tl }
10011          { groups }
10012       }
10013      {
10014        \prop_put:cnV { \c__keys_info_root_tl \l_keys_path_tl }
10015          { groups } \l__keys_groups_clist
10016      }
10017   }
```

(*End definition for* `\__keys_groups_set:n.`)

`\__keys_initialise:n` A set up for initialisation: just run the code if it exists.

```
10018 \cs_new_protected:Npn \__keys_initialise:n #1
10019   {
10020     \cs_if_exist_use:cT { \c__keys_code_root_tl \l_keys_path_tl } { {#1} }
10021   }
```

(*End definition for* `\__keys_initialise:n.`)

`\__keys_meta_make:n`
`\__keys_meta_make:nn`

To create a meta-key, simply set up to pass data through.

```
10022 \cs_new_protected:Npn \__keys_meta_make:n #1
10023   {
10024     \__keys_cmd_set:Vo \l_keys_path_tl
10025       {
10026         \exp_after:wN \keys_set:nn
10027         \exp_after:wN { \l__keys_module_tl } {#1}
10028       }
10029   }
10030 \cs_new_protected:Npn \__keys_meta_make:nn #1#2
10031   { \__keys_cmd_set:Vn \l_keys_path_tl { \keys_set:nn {#1} {#2} } }
```

(End definition for `\__keys_meta_make:n` *and* `\__keys_meta_make:nn`.*)*

`\__keys_undefine:`

Undefining a key has to be done without `\cs_undefine:c` as that function acts globally.

```
10032 \cs_new_protected:Npn \__keys_undefine:
10033   {
10034     \clist_map_inline:nn
10035       { code , default , groups , info , validate }
10036       {
10037         \cs_set_eq:cN
10038           { \tl_use:c { c__keys_ ##1 _root_tl } \l_keys_path_tl }
10039           \tex_undefined:D
10040       }
10041   }
```

(End definition for `\__keys_undefine:`.*)*

`\__keys_value_requirement:nn`
`\__keys_validate_forbidden:`
`\__keys_validate_required:`
`\__keys_validate_cleanup:w`

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.

```
10042 \cs_new_protected:Npn \__keys_value_requirement:nn #1#2
10043   {
10044     \str_case:nnF {#2}
10045       {
10046         { true }
10047         {
10048           \cs_set_eq:cc
10049             { \c__keys_validate_root_tl \l_keys_path_tl }
10050             { __keys_validate_ #1 : }
10051         }
10052         { false }
10053         {
10054           \cs_if_eq:ccT
10055             { \c__keys_validate_root_tl \l_keys_path_tl }
10056             { __keys_validate_ #1 : }
10057             {
10058               \cs_set_eq:cN
10059                 { \c__keys_validate_root_tl \l_keys_path_tl }
10060                 \tex_undefined:D
10061             }
10062         }
10063       }
```

```
10064            {
10065              \__msg_kernel_error:nnx { kernel } { property-boolean-values-only }
10066                { .value_ #1 :n }
10067            }
10068    }
10069  \cs_new_protected:Npn \__keys_validate_forbidden:
10070    {
10071      \bool_if:NF \l__keys_no_value_bool
10072        {
10073          \__msg_kernel_error:nnxx { kernel } { value-forbidden }
10074            { \l_keys_path_tl } { \l_keys_value_tl }
10075          \__keys_validate_cleanup:w
10076        }
10077    }
10078  \cs_new_protected:Npn \__keys_validate_required:
10079    {
10080      \bool_if:NT \l__keys_no_value_bool
10081        {
10082          \__msg_kernel_error:nnx { kernel } { value-required }
10083            { \l_keys_path_tl }
10084          \__keys_validate_cleanup:w
10085        }
10086    }
10087  \cs_new_protected:Npn \__keys_validate_cleanup:w #1 \cs_end: #2#3 { }
```

(*End definition for* __keys_value_requirement:nn *and others.*)

__keys_variable_set:NnnN   Setting a variable takes the type and scope separately so that it is easy to make a new
__keys_variable_set:cnnN   variable if needed.

```
10088  \cs_new_protected:Npn \__keys_variable_set:NnnN #1#2#3#4
10089    {
10090      \use:c { #2_if_exist:NF } #1 { \use:c { #2 _new:N } #1 }
10091      \__keys_cmd_set:nx { \l_keys_path_tl }
10092        {
10093          \exp_not:c { #2 _ #3 set:N #4 }
10094          \exp_not:N #1
10095          \exp_not:n   { {##1} }
10096        }
10097    }
10098  \cs_generate_variant:Nn \__keys_variable_set:NnnN { c }
```

(*End definition for* __keys_variable_set:NnnN.)

19.5   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   One function for this.
.bool_set:c
.bool_gset:N   `10099  \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:N } #1`
.bool_gset:c

```
10100      { \__keys_bool_set:Nn #1 { } }
10101  \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:c } #1
10102      { \__keys_bool_set:cn {#1} { } }
10103  \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:N } #1
10104      { \__keys_bool_set:Nn #1 { g } }
10105  \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:c } #1
10106      { \__keys_bool_set:cn {#1} { g } }
```

(*End definition for* .bool_set:N *and* .bool_gset:N. *These functions are documented on page* *162*.)

.bool_set_inverse:N One function for this.
.bool_set_inverse:c
.bool_gset_inverse:N
.bool_gset_inverse:c

```
10107  \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:N } #1
10108      { \__keys_bool_set_inverse:Nn #1 { } }
10109  \cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:c } #1
10110      { \__keys_bool_set_inverse:cn {#1} { } }
10111  \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:N } #1
10112      { \__keys_bool_set_inverse:Nn #1 { g } }
10113  \cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:c } #1
10114      { \__keys_bool_set_inverse:cn {#1} { g } }
```

(*End definition for* .bool_set_inverse:N *and* .bool_gset_inverse:N. *These functions are documented on page* *162*.)

.choice: Making a choice is handled internally, as it is also needed by .generate_choices:n.

```
10115  \cs_new_protected:cpn { \c__keys_props_root_tl .choice: }
10116      { \__keys_choice_make: }
```

(*End definition for* .choice:. *This function is documented on page* *162*.)

.choices:nn For auto-generation of a series of mutually-exclusive choices. Here, #1 will consist of two
.choices:Vn separate arguments, hence the slightly odd-looking implementation.
.choices:on
.choices:xn

```
10117  \cs_new_protected:cpn { \c__keys_props_root_tl .choices:nn } #1
10118      { \__keys_choices_make:nn #1 }
10119  \cs_new_protected:cpn { \c__keys_props_root_tl .choices:Vn } #1
10120      { \exp_args:NV \__keys_choices_make:nn #1 }
10121  \cs_new_protected:cpn { \c__keys_props_root_tl .choices:on } #1
10122      { \exp_args:No \__keys_choices_make:nn #1 }
10123  \cs_new_protected:cpn { \c__keys_props_root_tl .choices:xn } #1
10124      { \exp_args:Nx \__keys_choices_make:nn #1 }
```

(*End definition for* .choices:nn. *This function is documented on page* *162*.)

.code:n Creating code is simply a case of passing through to the underlying set function.

```
10125  \cs_new_protected:cpn { \c__keys_props_root_tl .code:n } #1
10126      { \__keys_cmd_set:nn { \l_keys_path_tl } {#1} }
```

(*End definition for* .code:n. *This function is documented on page* *162*.)

.clist_set:N
.clist_set:c
.clist_gset:N
.clist_gset:c

```
10127  \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:N } #1
10128      { \__keys_variable_set:NnnN #1 { clist } { } n }
10129  \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:c } #1
10130      { \__keys_variable_set:cnnN {#1} { clist } { } n }
10131  \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:N } #1
10132      { \__keys_variable_set:NnnN #1 { clist } { g } n }
10133  \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:c } #1
10134      { \__keys_variable_set:cnnN {#1} { clist } { g } n }
```

(End definition for `.clist_set:N` *and* `.clist_gset:N`. *These functions are documented on page [162](#).)*

`.default:n` Expansion is left to the internal functions.
`.default:V`
`.default:o` 10135 `\cs_new_protected:cpn { \c__keys_props_root_tl .default:n } #1`
`.default:x` 10136 `  { \__keys_default_set:n {#1} }`
             10137 `\cs_new_protected:cpn { \c__keys_props_root_tl .default:V } #1`
             10138 `  { \exp_args:NV \__keys_default_set:n #1 }`
             10139 `\cs_new_protected:cpn { \c__keys_props_root_tl .default:o } #1`
             10140 `  { \exp_args:No \__keys_default_set:n {#1} }`
             10141 `\cs_new_protected:cpn { \c__keys_props_root_tl .default:x } #1`
             10142 `  { \exp_args:Nx \__keys_default_set:n {#1} }`

(End definition for `.default:n`. *This function is documented on page [163](#).)*

`.dim_set:N`   Setting a variable is very easy: just pass the data along.
`.dim_set:c`
`.dim_gset:N` 10143 `\cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:N } #1`
`.dim_gset:c` 10144 `  { \__keys_variable_set:NnnN #1 { dim } { } n }`
             10145 `\cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:c } #1`
             10146 `  { \__keys_variable_set:cnnN {#1} { dim } { } n }`
             10147 `\cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:N } #1`
             10148 `  { \__keys_variable_set:NnnN #1 { dim } { g } n }`
             10149 `\cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1`
             10150 `  { \__keys_variable_set:cnnN {#1} { dim } { g } n }`

(End definition for `.dim_set:N` *and* `.dim_gset:N`. *These functions are documented on page [163](#).)*

`.fp_set:N`   Setting a variable is very easy: just pass the data along.
`.fp_set:c`
`.fp_gset:N` 10151 `\cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:N } #1`
`.fp_gset:c` 10152 `  { \__keys_variable_set:NnnN #1 { fp } { } n }`
            10153 `\cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:c } #1`
            10154 `  { \__keys_variable_set:cnnN {#1} { fp } { } n }`
            10155 `\cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:N } #1`
            10156 `  { \__keys_variable_set:NnnN #1 { fp } { g } n }`
            10157 `\cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1`
            10158 `  { \__keys_variable_set:cnnN {#1} { fp } { g } n }`

(End definition for `.fp_set:N` *and* `.fp_gset:N`. *These functions are documented on page [163](#).)*

`.groups:n`   A single property to create groups of keys.

            10159 `\cs_new_protected:cpn { \c__keys_props_root_tl .groups:n } #1`
            10160 `  { \__keys_groups_set:n {#1} }`

(End definition for `.groups:n`. *This function is documented on page [163](#).)*

`.initial:n` The standard hand-off approach.
`.initial:V`
`.initial:o` 10161 `\cs_new_protected:cpn { \c__keys_props_root_tl .initial:n } #1`
`.initial:x` 10162 `  { \__keys_initialise:n {#1} }`
            10163 `\cs_new_protected:cpn { \c__keys_props_root_tl .initial:V } #1`
            10164 `  { \exp_args:NV \__keys_initialise:n #1 }`
            10165 `\cs_new_protected:cpn { \c__keys_props_root_tl .initial:o } #1`
            10166 `  { \exp_args:No \__keys_initialise:n {#1} }`
            10167 `\cs_new_protected:cpn { \c__keys_props_root_tl .initial:x } #1`
            10168 `  { \exp_args:Nx \__keys_initialise:n {#1} }`

(End definition for `.initial:n`. *This function is documented on page [163](#).)*

.int_set:N Setting a variable is very easy: just pass the data along.
.int_set:c
.int_gset:N
.int_gset:c

```
10169 \cs_new_protected:cpn { \c__keys_props_root_tl .int_set:N } #1
10170    { \__keys_variable_set:NnnN #1 { int } { } n }
10171 \cs_new_protected:cpn { \c__keys_props_root_tl .int_set:c } #1
10172    { \__keys_variable_set:cnnN {#1} { int } { } n }
10173 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:N } #1
10174    { \__keys_variable_set:NnnN #1 { int } { g } n }
10175 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:c } #1
10176    { \__keys_variable_set:cnnN {#1} { int } { g } n }
```

(*End definition for* .int_set:N *and* .int_gset:N*. These functions are documented on page 163.*)

.meta:n Making a meta is handled internally.

```
10177 \cs_new_protected:cpn { \c__keys_props_root_tl .meta:n } #1
10178    { \__keys_meta_make:n {#1} }
```

(*End definition for* .meta:n*. This function is documented on page 163.*)

.meta:nn Meta with path: potentially lots of variants, but for the moment no so many defined.

```
10179 \cs_new_protected:cpn { \c__keys_props_root_tl .meta:nn } #1
10180    { \__keys_meta_make:nn #1 }
```

(*End definition for* .meta:nn*. This function is documented on page 164.*)

.multichoice: The same idea as .choice: and .choices:nn, but where more than one choice is allowed.
.multichoices:nn
.multichoices:Vn
.multichoices:on
.multichoices:xn

```
10181 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoice: }
10182    { \__keys_multichoice_make: }
10183 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:nn } #1
10184    { \__keys_multichoices_make:nn #1 }
10185 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:Vn } #1
10186    { \exp_args:NV \__keys_multichoices_make:nn #1 }
10187 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:on } #1
10188    { \exp_args:No \__keys_multichoices_make:nn #1 }
10189 \cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:xn } #1
10190    { \exp_args:Nx \__keys_multichoices_make:nn #1 }
```

(*End definition for* .multichoice: *and* .multichoices:nn*. These functions are documented on page 164.*)

.skip_set:N Setting a variable is very easy: just pass the data along.
.skip_set:c
.skip_gset:N
.skip_gset:c

```
10191 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:N } #1
10192    { \__keys_variable_set:NnnN #1 { skip } { } n }
10193 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:c } #1
10194    { \__keys_variable_set:cnnN {#1} { skip } { } n }
10195 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:N } #1
10196    { \__keys_variable_set:NnnN #1 { skip } { g } n }
10197 \cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:c } #1
10198    { \__keys_variable_set:cnnN {#1} { skip } { g } n }
```

(*End definition for* .skip_set:N *and* .skip_gset:N*. These functions are documented on page 164.*)

.tl_set:N   Setting a variable is very easy: just pass the data along.
.tl_set:c
.tl_gset:N 10199 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:N } #1
.tl_gset:c 10200   { __keys_variable_set:NnnN #1 { tl } { } n }
.tl_set_x:N 10201 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:c } #1
.tl_set_x:c 10202   { __keys_variable_set:cnnN {#1} { tl } { } n }
.tl_gset_x:N 10203 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:N } #1
.tl_gset_x:c 10204   { __keys_variable_set:NnnN #1 { tl } { } x }
          10205 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:c } #1
          10206   { __keys_variable_set:cnnN {#1} { tl } { } x }
          10207 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:N } #1
          10208   { __keys_variable_set:NnnN #1 { tl } { g } n }
          10209 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:c } #1
          10210   { __keys_variable_set:cnnN {#1} { tl } { g } n }
          10211 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:N } #1
          10212   { __keys_variable_set:NnnN #1 { tl } { g } x }
          10213 \cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:c } #1
          10214   { __keys_variable_set:cnnN {#1} { tl } { g } x }

(*End definition for* .tl_set:N *and others. These functions are documented on page 164.*)

.undefine: Another simple wrapper.

          10215 \cs_new_protected:cpn { \c__keys_props_root_tl .undefine: }
          10216   { __keys_undefine: }

(*End definition for* .undefine:. *This function is documented on page 164.*)

.value_forbidden:n These are very similar, so both call the same function.
.value_required:n
          10217 \cs_new_protected:cpn { \c__keys_props_root_tl .value_forbidden:n } #1
          10218   { __keys_value_requirement:nn { forbidden } {#1} }
          10219 \cs_new_protected:cpn { \c__keys_props_root_tl .value_required:n } #1
          10220   { __keys_value_requirement:nn { required } {#1} }

(*End definition for* .value_forbidden:n *and* .value_required:n. *These functions are documented on page 164.*)

19.6   Setting keys

\keys_set:nn   A simple wrapper again.
\keys_set:nV
\keys_set:nv   10221 \cs_new_protected:Npn \keys_set:nn
\keys_set:no   10222   { __keys_set:onn { \l__keys_module_tl } }
__keys_set:nnn 10223 \cs_new_protected:Npn __keys_set:nnn #1#2#3
__keys_set:onn 10224   {
          10225   \tl_set:Nx \l__keys_module_tl { __keys_remove_spaces:n {#2} }
          10226   \keyval_parse:NNn __keys_set:n __keys_set:nn {#3}
          10227   \tl_set:Nn \l__keys_module_tl {#1}
          10228   }
          10229 \cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
          10230 \cs_generate_variant:Nn __keys_set:nnn { o }

(*End definition for* \keys_set:nn *and* __keys_set:nnn. *These functions are documented on page 167.*)

515

Setting known keys simply means setting the appropriate flag, then running the standard code. To allow for nested setting, any existing value of `\l__keys_unused_clist` is saved on the stack and reset afterwards. Note that for speed/simplicity reasons we use a `tl` operation to set the `clist` here!

```
10231 \cs_new_protected:Npn \keys_set_known:nnN
10232   { \__keys_set_known:onnN \l__keys_unused_clist }
10233 \cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
10234 \cs_new_protected:Npn \__keys_set_known:nnnN #1#2#3#4
10235   {
10236     \clist_clear:N \l__keys_unused_clist
10237     \keys_set_known:nn {#2} {#3}
10238     \tl_set:Nx #4 { \exp_not:o { \l__keys_unused_clist } }
10239     \tl_set:Nn \l__keys_unused_clist {#1}
10240   }
10241 \cs_generate_variant:Nn \__keys_set_known:nnnN { o }
10242 \cs_new_protected:Npn \keys_set_known:nn #1#2
10243   {
10244     \bool_set_true:N \l__keys_only_known_bool
10245     \keys_set:nn {#1} {#2}
10246     \bool_set_false:N \l__keys_only_known_bool
10247   }
10248 \cs_generate_variant:Nn \keys_set_known:nn { nV , nv , no }
```

(*End definition for* `\keys_set_known:nnN`, `\__keys_set_known:nnnN`, *and* `\keys_set_known:nn`. *These functions are documented on page 168.*)

The idea of setting keys in a selective manner again uses flags wrapped around the basic code. The comments on `\keys_set_known:nnN` also apply here.

```
10249 \cs_new_protected:Npn \keys_set_filter:nnnN
10250   { \__keys_set_filter:onnnN \l__keys_unused_clist }
10251 \cs_generate_variant:Nn \keys_set_filter:nnnN { nnV , nnv , nno }
10252 \cs_new_protected:Npn \__keys_set_filter:nnnnN #1#2#3#4#5
10253   {
10254     \clist_clear:N \l__keys_unused_clist
10255     \keys_set_filter:nnn {#2} {#3} {#4}
10256     \tl_set:Nx #5 { \exp_not:o { \l__keys_unused_clist } }
10257     \tl_set:Nn \l__keys_unused_clist {#1}
10258   }
10259 \cs_generate_variant:Nn \__keys_set_filter:nnnnN { o }
10260 \cs_new_protected:Npn \keys_set_filter:nnn #1#2#3
10261   {
10262     \bool_set_true:N \l__keys_selective_bool
10263     \bool_set_true:N \l__keys_filtered_bool
10264     \seq_set_from_clist:Nn \l__keys_selective_seq {#2}
10265     \keys_set:nn {#1} {#3}
10266     \bool_set_false:N \l__keys_selective_bool
10267   }
10268 \cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }
10269 \cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
10270   {
10271     \bool_set_true:N \l__keys_selective_bool
10272     \bool_set_false:N \l__keys_filtered_bool
10273     \seq_set_from_clist:Nn \l__keys_selective_seq {#2}
10274     \keys_set:nn {#1} {#3}
```

516

```
10275        \bool_set_false:N \l__keys_selective_bool
10276      }
10277    \cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }
```

(*End definition for* `\keys_set_filter:nnnN` *and others. These functions are documented on page* *169.*)

`\__keys_set:n`
`\__keys_set:nn`
`\__keys_set_aux:nnn`
`\__keys_set_aux:onn`
`\__keys_find_key_module:w`
`\__keys_set_aux:`
`\__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 the a value is required or forbidden. If everything passes, move on to execute the code.

```
10278  \cs_new_protected:Npn \__keys_set:n #1
10279    {
10280      \bool_set_true:N \l__keys_no_value_bool
10281      \__keys_set_aux:onn \l__keys_module_tl {#1} { }
10282    }
10283  \cs_new_protected:Npn \__keys_set:nn #1#2
10284    {
10285      \bool_set_false:N \l__keys_no_value_bool
10286      \__keys_set_aux:onn \l__keys_module_tl {#1} {#2}
10287    }
```

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.

```
10288  \cs_new_protected:Npn \__keys_set_aux:nnn #1#2#3
10289    {
10290      \tl_set:Nx \l_keys_path_tl { #1 / \__keys_remove_spaces:n {#2} }
10291      \tl_clear:N \l__keys_module_tl
10292      \exp_after:wN \__keys_find_key_module:w \l_keys_path_tl / \q_stop
10293      \__keys_value_or_default:n {#3}
10294      \bool_if:NTF \l__keys_selective_bool
10295        { \__keys_set_selective: }
10296        { \__keys_execute: }
10297      \tl_set:Nn \l__keys_module_tl {#1}
10298    }
10299  \cs_generate_variant:Nn \__keys_set_aux:nnn { o }
10300  \cs_new_protected:Npn \__keys_find_key_module:w #1 / #2 \q_stop
10301    {
10302      \tl_if_blank:nTF {#2}
10303        { \tl_set:Nn \l_keys_key_tl {#1} }
10304        {
10305          \tl_put_right:Nx \l__keys_module_tl
10306            {
10307              \tl_if_empty:NF \l__keys_module_tl { / }
10308              #1
10309            }
10310          \__keys_find_key_module:w #2 \q_stop
10311        }
10312    }
```

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.

```
10313  \cs_new_protected:Npn \__keys_set_selective:
```

```
10314        {
10315          \prop_if_exist:cTF { \c__keys_info_root_tl \l_keys_path_tl }
10316            {
10317              \prop_get:cnNTF { \c__keys_info_root_tl \l_keys_path_tl }
10318                { groups } \l__keys_groups_clist
10319                { \__keys_check_groups: }
10320                {
10321                  \bool_if:NTF \l__keys_filtered_bool
10322                    { \__keys_execute: }
10323                    { \__keys_store_unused: }
10324                }
10325            }
10326            {
10327              \bool_if:NTF \l__keys_filtered_bool
10328                { \__keys_execute: }
10329                { \__keys_store_unused: }
10330            }
10331        }
```

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.

```
10332  \cs_new_protected:Npn \__keys_check_groups:
10333    {
10334      \bool_set_false:N \l__keys_tmp_bool
10335      \seq_map_inline:Nn \l__keys_selective_seq
10336        {
10337          \clist_map_inline:Nn \l__keys_groups_clist
10338            {
10339              \str_if_eq:nnT {##1} {####1}
10340                {
10341                  \bool_set_true:N \l__keys_tmp_bool
10342                  \clist_map_break:n { \seq_map_break: }
10343                }
10344            }
10345        }
10346      \bool_if:NTF \l__keys_tmp_bool
10347        {
10348          \bool_if:NTF \l__keys_filtered_bool
10349            { \__keys_store_unused: }
10350            { \__keys_execute: }
10351        }
10352        {
10353          \bool_if:NTF \l__keys_filtered_bool
10354            { \__keys_execute: }
10355            { \__keys_store_unused: }
10356        }
10357    }
```

(*End definition for* `\__keys_set:n` *and others.*)

`\__keys_value_or_default:n`   If a value is given, return it as `#1`, otherwise send a default if available.

```
10358  \cs_new_protected:Npn \__keys_value_or_default:n #1
10359    {
10360      \bool_if:NTF \l__keys_no_value_bool
```

```
10361        {
10362          \cs_if_exist:cTF { \c__keys_default_root_tl \l_keys_path_tl }
10363            {
10364              \tl_set_eq:Nc
10365                \l_keys_value_tl
10366                { \c__keys_default_root_tl \l_keys_path_tl }
10367            }
10368            { \tl_clear:N \l_keys_value_tl }
10369        }
10370        { \tl_set:Nn \l_keys_value_tl {#1} }
10371    }
```

(*End definition for* __keys_value_or_default:n.)

__keys_execute:
__keys_execute_unknown:
__keys_execute:nn
__keys_store_unused:

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.

```
10372 \cs_new_protected:Npn \__keys_execute:
10373    {
10374      \cs_if_exist:cTF { \c__keys_code_root_tl \l_keys_path_tl }
10375        {
10376          \cs_if_exist_use:c { \c__keys_validate_root_tl \l_keys_path_tl }
10377          \cs:w \c__keys_code_root_tl \l_keys_path_tl \exp_after:wN \cs_end:
10378            \exp_after:wN { \l_keys_value_tl }
10379        }
10380        { \__keys_execute_unknown: }
10381    }
10382 \cs_new_protected:Npn \__keys_execute_unknown:
10383    {
10384      \bool_if:NTF \l__keys_only_known_bool
10385        { \__keys_store_unused: }
10386        {
10387          \cs_if_exist:cTF { \c__keys_code_root_tl \l__keys_module_tl / unknown }
10388            {
10389              \cs:w \c__keys_code_root_tl \l__keys_module_tl / unknown \exp_after:wN
10390                \cs_end: \exp_after:wN { \l_keys_value_tl }
10391            }
10392            {
10393              \__msg_kernel_error:nnxx { kernel } { key-unknown }
10394                { \l_keys_path_tl } { \l__keys_module_tl }
10395            }
10396        }
10397    }
10398 \cs_new:Npn \__keys_execute:nn #1#2
10399    {
10400      \cs_if_exist:cTF { \c__keys_code_root_tl #1 }
10401        {
10402          \exp_args:Nc \exp_args:No { \c__keys_code_root_tl #1 }
10403            \l_keys_value_tl
10404        }
10405        {#2}
10406    }
10407 \cs_new_protected:Npn \__keys_store_unused:
```

```
10408    {
10409      \clist_put_right:Nx \l__keys_unused_clist
10410        {
10411          \exp_not:o \l_keys_key_tl
10412          \bool_if:NF \l__keys_no_value_bool
10413            { = { \exp_not:o \l_keys_value_tl } }
10414        }
10415    }
```

(*End definition for* __keys_execute: *and others.*)

__keys_choice_find:n
__keys_multichoice_find:n

Executing a choice has two parts. First, try the choice given, then if that fails call the unknown key. That will exist, 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.

```
10416 \cs_new:Npn \__keys_choice_find:n #1
10417    {
10418      \__keys_execute:nn { \l_keys_path_tl / \__keys_remove_spaces:n {#1} }
10419        { \__keys_execute:nn { \l_keys_path_tl / unknown } { } }
10420    }
10421 \cs_new:Npn \__keys_multichoice_find:n #1
10422    { \clist_map_function:nN {#1} \__keys_choice_find:n }
```

(*End definition for* __keys_choice_find:n *and* __keys_multichoice_find:n.)

19.7 Utilities

__keys_remove_spaces:n
__keys_remove_spaces:w

Removes all spaces from the input which is detokenized as a result. This function has the same effect as \zap@space in LaTeX 2_ε after applying \tl_to_str:n. It is set up to be fast as the use case here is tightly defined. The ? is only there to allow for a space after \use_none:nn responsible for ending the loop.

```
10423 \cs_new:Npn \__keys_remove_spaces:n #1
10424    {
10425      \exp_after:wN \__keys_remove_spaces:w \tl_to_str:n {#1}
10426      \use_none:nn ? ~
10427    }
10428 \cs_new:Npn \__keys_remove_spaces:w #1 ~
10429    { #1 \__keys_remove_spaces:w }
```

(*End definition for* __keys_remove_spaces:n *and* __keys_remove_spaces:w.)

\keys_if_exist_p:nn
\keys_if_exist:nnTF

A utility for others to see if a key exists.

```
10430 \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
10431    {
10432      \cs_if_exist:cTF
10433        { \c__keys_code_root_tl \__keys_remove_spaces:n { #1 / #2 } }
10434        { \prg_return_true: }
10435        { \prg_return_false: }
10436    }
```

(*End definition for* \keys_if_exist:nnTF. *This function is documented on page 170.*)

\keys_if_choice_exist_p:nnn
\keys_if_choice_exist:nnn*TF*

Just an alternative view on \keys_if_exist:nnTF.

```
10437 \prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3
10438   { p , T , F , TF }
10439   {
10440     \cs_if_exist:cTF
10441       { \c__keys_code_root_tl \__keys_remove_spaces:n { #1 / #2 / #3 } }
10442       { \prg_return_true: }
10443       { \prg_return_false: }
10444   }
```

(*End definition for* \keys_if_choice_exist:nnnTF. *This function is documented on page 170.*)

\keys_show:nn
__keys_show:NN

To show a key, test for its existence to issue the correct message (same message, but with a t or f argument, then build the control sequences which contain the code and other information about the key, call an intermediate auxiliary which constructs the code that will be displayed to the terminal, and finally conclude with __msg_show_wrap:n.

```
10445 \cs_new_protected:Npn \keys_show:nn #1#2
10446   {
10447     \keys_if_exist:nnTF {#1} {#2}
10448       {
10449         \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-key }
10450           { \__keys_remove_spaces:n { #1 / #2 } } { t } { } { }
10451         \exp_args:Ncc \__keys_show:NN
10452           { \c__keys_code_root_tl \__keys_remove_spaces:n { #1 / #2 } }
10453           { \c__keys_info_root_tl \__keys_remove_spaces:n { #1 / #2 } }
10454       }
10455       {
10456         \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-key }
10457           { \__keys_remove_spaces:n { #1 / #2 } } { f } { } { }
10458         \__msg_show_wrap:n { }
10459       }
10460   }
10461 \cs_new_protected:Npn \__keys_show:NN #1#2
10462   {
10463     \use:x
10464       {
10465         \__msg_show_wrap:n
10466           {
10467             \exp_not:N \__msg_show_item_unbraced:nn { code }
10468               { \token_get_replacement_spec:N #1 }
10469             \exp_not:n
10470               { \prop_map_function:NN #2 \__msg_show_item_unbraced:nn }
10471           }
10472       }
10473   }
```

(*End definition for* \keys_show:nn *and* __keys_show:NN. *These functions are documented on page 170.*)

19.8 Messages

For when there is a need to complain.

```
10474 \__msg_kernel_new:nnnn { kernel } { boolean-values-only }
10475   { Key~'#1'~accepts~boolean~values~only. }
10476   { The~key~'#1'~only~accepts~the~values~'true'~and~'false'. }
```

521

```
10477 \__msg_kernel_new:nnnn { kernel } { key-choice-unknown }
10478   { Key~'#1'~accepts~only~a~fixed~set~of~choices. }
10479   {
10480     The~key~'#1'~only~accepts~predefined~values,~
10481     and~'#2'~is~not~one~of~these.
10482   }
10483 \__msg_kernel_new:nnnn { kernel } { key-no-property }
10484   { No~property~given~in~definition~of~key~'#1'. }
10485   {
10486     \c__msg_coding_error_text_tl
10487     Inside~\keys_define:nn   each~key~name~
10488     needs~a~property:  \\ \\
10489     \iow_indent:n { #1 .<property> } \\ \\
10490     LaTeX~did~not~find~a~'.'~to~indicate~the~start~of~a~property.
10491   }
10492 \__msg_kernel_new:nnnn { kernel } { key-unknown }
10493   { The~key~'#1'~is~unknown~and~is~being~ignored. }
10494   {
10495     The~module~'#2'~does~not~have~a~key~called~'#1'.\\
10496     Check~that~you~have~spelled~the~key~name~correctly.
10497   }
10498 \__msg_kernel_new:nnnn { kernel } { nested-choice-key }
10499   { Attempt~to~define~'#1'~as~a~nested~choice~key. }
10500   {
10501     The~key~'#1'~cannot~be~defined~as~a~choice~as~the~parent~key~'#2'~is~
10502     itself~a~choice.
10503   }
10504 \__msg_kernel_new:nnnn { kernel } { property-boolean-values-only }
10505   { The~property~'#1'~accepts~boolean~values~only. }
10506   {
10507     \c__msg_coding_error_text_tl
10508     The~property~'#1'~only~accepts~the~values~'true'~and~'false'.
10509   }
10510 \__msg_kernel_new:nnnn { kernel } { property-requires-value }
10511   { The~property~'#1'~requires~a~value. }
10512   {
10513     \c__msg_coding_error_text_tl
10514     LaTeX~was~asked~to~set~property~'#1'~for~key~'#2'.\\
10515     No~value~was~given~for~the~property,~and~one~is~required.
10516   }
10517 \__msg_kernel_new:nnnn { kernel } { property-unknown }
10518   { The~key~property~'#1'~is~unknown. }
10519   {
10520     \c__msg_coding_error_text_tl
10521     LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
10522     this~property~is~not~defined.
10523   }
10524 \__msg_kernel_new:nnnn { kernel } { value-forbidden }
10525   { The~key~'#1'~does~not~take~a~value. }
10526   {
10527     The~key~'#1'~should~be~given~without~a~value.\\
10528     The~value~'#2'~was~present:~the~key~will~be~ignored.
10529   }
10530 \__msg_kernel_new:nnnn { kernel } { value-required }
```

```
10531      { The~key~'#1'~requires~a~value. }
10532      {
10533        The~key~'#1'~must~have~a~value.\\
10534        No~value~was~present:~the~key~will~be~ignored.
10535      }
10536  \__msg_kernel_new:nnn { kernel } { show-key }
10537      {
10538        The~key~#1~
10539        \str_if_eq:nnTF {#2} { t }
10540          { has~the~properties: }
10541          { is~undefined. }
10542      }
```

19.9 Deprecated functions

.value_forbidden: Deprecated 2015-07-14.

.value_required:
```
10543 \cs_new_protected:cpn { \c__keys_props_root_tl .value_forbidden: }
10544    { \__keys_value_requirement:nn { forbidden } { true } }
10545 \cs_new_protected:cpn { \c__keys_props_root_tl .value_required: }
10546    { \__keys_value_requirement:nn { required } { true } }
```

(*End definition for* .value_forbidden: *and* .value_required:.)

```
10547 ⟨/initex | package⟩
```

20 l3file implementation

The following test files are used for this code: m3file001.

```
10548 ⟨*initex | package⟩
```

```
10549 ⟨@@=file⟩
```

20.1 File operations

\g_file_current_name_tl   The name of the current file should be available at all times. For the format the file name needs to be picked up at the start of the file. In LaTeX 2ε package mode the current file name is collected from \@currname.

```
10550 \tl_new:N \g_file_current_name_tl
10551 ⟨*initex⟩
10552 \tex_everyjob:D \exp_after:wN
10553    {
10554      \tex_the:D \tex_everyjob:D
10555      \tl_gset:Nx \g_file_current_name_tl { \tex_jobname:D }
10556    }
10557 ⟨/initex⟩
10558 ⟨*package⟩
10559 \cs_if_exist:NT \@currname
10560    { \tl_gset_eq:NN \g_file_current_name_tl \@currname }
10561 ⟨/package⟩
```

(*End definition for* \g_file_current_name_tl. *This variable is documented on page 172.*)

\g__file_stack_seq   The input list of files is stored as a sequence stack.

```
10562 \seq_new:N \g__file_stack_seq
```

(*End 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! In format mode, this is done at the very start of the TEX run. In package mode we will eventually copy the contents of \@filelist.

```
10563 \seq_new:N \g__file_record_seq
10564 ⟨*initex⟩
10565 \tex_everyjob:D \exp_after:wN
10566   {
10567     \tex_the:D \tex_everyjob:D
10568     \seq_gput_right:NV \g__file_record_seq \g_file_current_name_tl
10569   }
10570 ⟨/initex⟩
```

(*End definition for* \g__file_record_seq.)

\l__file_internal_tl   Used as a short-term scratch variable. It may be possible to reuse \l__file_internal_-
name_tl there.

```
10571 \tl_new:N \l__file_internal_tl
```

(*End definition for* \l__file_internal_tl.)

\l__file_internal_name_tl   Used to return the fully-qualified name of a file.

```
10572 \tl_new:N \l__file_internal_name_tl
```

(*End definition for* \l__file_internal_name_tl.)

\l__file_search_path_seq   The current search path.

```
10573 \seq_new:N \l__file_search_path_seq
```

(*End definition for* \l__file_search_path_seq.)

\l__file_saved_search_path_seq   The current search path has to be saved for package use.

```
10574 ⟨*package⟩
10575 \seq_new:N \l__file_saved_search_path_seq
10576 ⟨/package⟩
```

(*End definition for* \l__file_saved_search_path_seq.)

\l__file_internal_seq   Scratch space for comma list conversion in package mode.

```
10577 ⟨*package⟩
10578 \seq_new:N \l__file_internal_seq
10579 ⟨/package⟩
```

(*End definition for* \l__file_internal_seq.)

__file_name_sanitize:nn   For converting a token list to a string where active characters are treated as strings
__file_name_sanitize_aux:n   from the start. The logic to the quoting normalisation is the same as used by lualatexquotejobname: check for balanced ", and assuming they balance strip all of them out before quoting the entire name if it contains spaces.

```
10580 \cs_new_protected:Npn \__file_name_sanitize:nn #1#2
10581   {
10582     \group_begin:
10583       \seq_map_inline:Nn \l_char_active_seq
```

524

```
10584            { \char_set:active:Npx ##1 { \cs_to_str:N ##1 } }
10585         \tl_set:Nx \l__file_internal_name_tl {#1}
10586         \tl_set:Nx \l__file_internal_name_tl
10587            { \tl_to_str:N \l__file_internal_name_tl }
10588         \int_compare:nNnTF
10589            {
10590              \int_mod:nn
10591                {
10592                  0 \tl_map_function:NN \l__file_internal_name_tl
10593                    \__file_name_sanitize_aux:n
10594                }
10595                \c_two
10596            }
10597            = \c_zero
10598            {
10599              \tl_remove_all:Nn \l__file_internal_name_tl { " }
10600              \tl_if_in:NnT \l__file_internal_name_tl { ~ }
10601                {
10602                  \tl_set:Nx \l__file_internal_name_tl
10603                    { " \exp_not:V \l__file_internal_name_tl " }
10604                }
10605            }
10606            {
10607              \__msg_kernel_error:nnx
10608                { kernel } { unbalanced-quote-in-filename }
10609                { \l__file_internal_name_tl }
10610            }
10611         \use:x
10612            {
10613              \group_end:
10614              \exp_not:n {#2} { \l__file_internal_name_tl }
10615            }
10616      }
10617   \cs_new:Npn \__file_name_sanitize_aux:n #1
10618      {
10619         \token_if_eq_charcode:NNT #1 "
10620            { + \c_one }
10621      }
```

(*End definition for* __file_name_sanitize:nn *and* __file_name_sanitize_aux:n.)

\file_add_path:nN   The way to test if a file exists is to try to open it: if it does not exist then TeX will
__file_add_path:nN   report end-of-file. For files which are in the current directory, this is straight-forward.
__file_add_path_search:nN   For other locations, a search has to be made looking at each potential path in turn. The
first location is of course treated as the correct one. If nothing is found, #2 is returned
empty.

```
10622   \cs_new_protected:Npn \file_add_path:nN #1
10623      { \__file_name_sanitize:nn {#1} { \__file_add_path:nN } }
10624   \cs_new_protected:Npn \__file_add_path:nN #1#2
10625      {
10626         \__ior_open:Nn \g__file_internal_ior {#1}
10627         \ior_if_eof:NTF \g__file_internal_ior
10628            { \__file_add_path_search:nN {#1} #2 }
10629            { \tl_set:Nn #2 {#1} }
```

```
10630          \ior_close:N \g__file_internal_ior
10631      }
10632  \cs_new_protected:Npn \__file_add_path_search:nN #1#2
10633      {
10634          \tl_set:Nn #2 { \q_no_value  }
```
⟨*package⟩
```
10636          \cs_if_exist:NT \input@path
10637            {
10638                \seq_set_eq:NN \l__file_saved_search_path_seq
10639                  \l__file_search_path_seq
10640                \seq_set_split:NnV \l__file_internal_seq { , } \input@path
10641                \seq_concat:NNN \l__file_search_path_seq
10642                  \l__file_search_path_seq \l__file_internal_seq
10643            }
```
⟨/package⟩
```
10645          \seq_map_inline:Nn \l__file_search_path_seq
10646            {
10647                \__ior_open:Nn \g__file_internal_ior { ##1 #1 }
10648                \ior_if_eof:NF \g__file_internal_ior
10649                  {
10650                      \tl_set:Nx #2 { ##1 #1 }
10651                      \seq_map_break:
10652                  }
10653            }
```
⟨*package⟩
```
10655          \cs_if_exist:NT \input@path
10656            {
10657                \seq_set_eq:NN \l__file_search_path_seq
10658                  \l__file_saved_search_path_seq
10659            }
```
⟨/package⟩
```
10661      }
```

(*End definition for* \file_add_path:nN *,* __file_add_path:nN *, and* __file_add_path_search:nN*. These functions are documented on page 172.*)

\file_if_exist:nTF   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 will contain something, whereas if the file was not located then the return value will be \q_no_value.

```
10662  \prg_new_protected_conditional:Npnn \file_if_exist:n #1 { T , F , TF }
10663      {
10664          \file_add_path:nN {#1} \l__file_internal_name_tl
10665          \quark_if_no_value:NTF \l__file_internal_name_tl
10666            { \prg_return_false: }
10667            { \prg_return_true: }
10668      }
```

(*End definition for* \file_if_exist:nTF*. This function is documented on page 172.*)

\file_input:n   Loading a file is done in a safe way, checking first that the file exists and loading only
__file_if_exist:nT   if it does. Push the file name on the \g__file_stack_seq, and add it to the file list,
__file_input:n __file_input:V   either \g__file_record_seq, or \@filelist in package mode.
__file_input_aux:n
__file_input_aux:o

```
10669  \cs_new_protected:Npn \file_input:n #1
10670      {
```

```
10671        \__file_if_exist:nT {#1}
10672          { \__file_input:V \l__file_internal_name_tl }
10673      }
```

This code is spun out as a separate function to encapsulate the error message into a easy-to-reuse form.

```
10674  \cs_new_protected:Npn \__file_if_exist:nT #1#2
10675    {
10676      \file_if_exist:nTF {#1}
10677        {#2}
10678        {
10679          \__file_name_sanitize:nn {#1}
10680            { \__msg_kernel_error:nnx { kernel } { file-not-found } }
10681        }
10682    }
10683  \cs_new_protected:Npn \__file_input:n #1
10684    {
10685      \tl_if_in:nnTF {#1} { . }
10686        { \__file_input_aux:n {#1} }
10687        { \__file_input_aux:o { \tl_to_str:n { #1 . tex } } }
10688    }
10689  \cs_generate_variant:Nn \__file_input:n { V }
10690  \cs_new_protected:Npn \__file_input_aux:n #1
10691    {
10692  ⟨*initex⟩
10693      \seq_gput_right:Nn \g__file_record_seq {#1}
10694  ⟨/initex⟩
10695  ⟨*package⟩
10696      \clist_if_exist:NTF \@filelist
10697        { \@addtofilelist {#1} }
10698        { \seq_gput_right:Nn \g__file_record_seq {#1} }
10699  ⟨/package⟩
10700      \seq_gpush:No \g__file_stack_seq \g_file_current_name_tl
10701      \tl_gset:Nn \g_file_current_name_tl {#1}
10702      \tex_input:D #1 \c_space_tl
10703      \seq_gpop:NN \g__file_stack_seq \l__file_internal_tl
10704      \tl_gset_eq:NN \g_file_current_name_tl \l__file_internal_tl
10705    }
10706  \cs_generate_variant:Nn \__file_input_aux:n { o }
```

(*End definition for* \file_input:n *and others. These functions are documented on page 172.*)

\file_path_include:n Wrapper functions to manage the search path.
\file_path_remove:n
__file_path_include:n
```
10707  \cs_new_protected:Npn \file_path_include:n #1
10708    { \__file_name_sanitize:nn {#1} { \__file_path_include:n } }
10709  \cs_new_protected:Npn \__file_path_include:n #1
10710    {
10711      \seq_if_in:NnF \l__file_search_path_seq {#1}
10712        { \seq_put_right:Nn \l__file_search_path_seq {#1} }
10713    }
10714  \cs_new_protected:Npn \file_path_remove:n #1
10715    {
10716      \__file_name_sanitize:nn {#1}
10717        { \seq_remove_all:Nn \l__file_search_path_seq }
10718    }
```

(*End definition for* \file_path_include:n *,* \file_path_remove:n *, and* __file_path_include:n *. These functions are documented on page* *172.*)

\file_list: 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 each file name into a string.

```
10719 \cs_new_protected:Npn \file_list:
10720   {
10721     \seq_set_eq:NN \l__file_internal_seq \g__file_record_seq
10722 ⟨*package⟩
10723     \clist_if_exist:NT \@filelist
10724       {
10725         \clist_map_inline:Nn \@filelist
10726           {
10727             \seq_put_right:No \l__file_internal_seq
10728               { \tl_to_str:n {##1} }
10729           }
10730       }
10731 ⟨/package⟩
10732     \seq_remove_duplicates:N \l__file_internal_seq
10733     \iow_log:n { *~File~List~* }
10734     \seq_map_inline:Nn \l__file_internal_seq { \iow_log:n {##1} }
10735     \iow_log:n { ************* }
10736   }
```

(*End definition for* \file_list:*. This function is documented on page* *173.*)

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.

```
10737 ⟨*package⟩
10738 \AtBeginDocument
10739   {
10740     \clist_map_inline:Nn \@filelist
10741       { \seq_gput_right:No \g__file_record_seq { \tl_to_str:n {#1} } }
10742   }
10743 ⟨/package⟩
```

20.2 Input operations

```
10744 ⟨@@=ior⟩
```

20.2.1 Variables and constants

\c_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.

```
10745 \cs_new_eq:NN \c_term_ior \c_sixteen
```

(*End definition for* \c_term_ior*. This variable is documented on page* *178.*)

\g__ior_streams_seq   A list of the currently-available input streams to be used as a stack. In format mode, all streams (from 0 to 15) are available, while the package requests streams to LaTeX 2_ε as they are needed (initially none are needed), so the starting point varies!

```
10746 \seq_new:N \g__ior_streams_seq
```

```
10747 ⟨*initex⟩
10748 \seq_gset_split:Nnn \g__ior_streams_seq { , }
10749    { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 }
10750 ⟨/initex⟩
```

(*End definition for* `\g__ior_streams_seq`.)

`\l__ior_stream_tl` Used to recover the raw stream number from the stack.

```
10751 \tl_new:N \l__ior_stream_tl
```

(*End 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 2ε 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 2ε mechanism it holds the value of the *last* stream allocated.

```
10752 \prop_new:N \g__ior_streams_prop
10753 ⟨*package⟩
10754 \int_step_inline:nnnn
10755    { \c_zero }
10756    { \c_one }
10757    {
10758       \cs_if_exist:NTF \normalend
10759          { \tex_count:D 38 \scan_stop: }
10760          {
10761             \tex_count:D 16 \scan_stop:
10762                \cs_if_exist:NT \loccount { - \c_one }
10763          }
10764    }
10765    {
10766       \prop_gput:Nnn \g__ior_streams_prop {#1} { Reserved~by~format }
10767    }
10768 ⟨/package⟩
```

(*End definition for* `\g__ior_streams_prop`.)

20.2.2 Stream management

`\ior_new:N` Reserving a new stream is done by defining the name as equal to using the terminal.
`\ior_new:c`
```
10769 \cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c_term_ior }
10770 \cs_generate_variant:Nn \ior_new:N { c }
```

(*End definition for* `\ior_new:N`. *This function is documented on page 173.*)

`\ior_open:Nn` Opening an input stream requires a bit of pre-processing. The file name is sanitized to
`\ior_open:cn` deal with active characters, before an auxiliary adds a path and checks that the file really
`\__ior_open_aux:Nn` exists. If those two tests pass, then pass the information on to the lower-level function which deals with streams.

```
10771 \cs_new_protected:Npn \ior_open:Nn #1#2
10772    { \__file_name_sanitize:nn {#2} { \__ior_open_aux:Nn #1 } }
10773 \cs_generate_variant:Nn \ior_open:Nn { c }
10774 \cs_new_protected:Npn \__ior_open_aux:Nn #1#2
```

```
10775        {
10776          \file_add_path:nN {#2} \l__file_internal_name_tl
10777          \quark_if_no_value:NTF \l__file_internal_name_tl
10778            { \__msg_kernel_error:nnx { kernel } { file-not-found } {#2} }
10779            { \__ior_open:No #1 \l__file_internal_name_tl }
10780        }
```

(*End definition for* \ior_open:Nn *and* __ior_open_aux:Nn. *These functions are documented on page*
173.)

\ior_open:Nn*TF*
\ior_open:cn*TF*
__ior_open_aux:Nn*TF*

Much the same idea for opening a read with a conditional, except the auxiliary function
does not issue an error if the file is not found.

```
10781 \prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
10782    { \__file_name_sanitize:nn {#2} { \__ior_open_aux:NnTF #1 } }
10783 \cs_generate_variant:Nn \ior_open:NnT  { c }
10784 \cs_generate_variant:Nn \ior_open:NnF  { c }
10785 \cs_generate_variant:Nn \ior_open:NnTF { c }
10786 \cs_new_protected:Npn \__ior_open_aux:NnTF #1#2
10787    {
10788      \file_add_path:nN {#2} \l__file_internal_name_tl
10789      \quark_if_no_value:NTF \l__file_internal_name_tl
10790        { \prg_return_false: }
10791        {
10792          \__ior_open:No #1 \l__file_internal_name_tl
10793          \prg_return_true:
10794        }
10795    }
```

(*End definition for* \ior_open:NnTF *and* __ior_open_aux:NnTF. *These functions are documented on*
page 173.)

__ior_new:N

In package mode, 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.

```
10796 ⟨*package⟩
10797 \exp_args:NNf \cs_new_protected:Npn \__ior_new:N
10798    { \exp_args:NNc \exp_after:wN \exp_stop_f: { newread } }
10799 ⟨/package⟩
```

(*End definition for* __ior_new:N.)

__ior_open:Nn
__ior_open:No
__ior_open_stream:Nn

The stream allocation itself uses the fact that there is a list of all of those available, so
allocation is simply a question of using the number at the top of the list. In package
mode, 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).

```
10800 \cs_new_protected:Npn \__ior_open:Nn #1#2
10801    {
10802      \ior_close:N #1
10803      \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
10804        { \__ior_open_stream:Nn #1 {#2} }
```

```
10805 ⟨*initex⟩
10806         { \__msg_kernel_fatal:nn { kernel } { input-streams-exhausted } }
10807 ⟨/initex⟩
10808 ⟨*package⟩
10809         {
10810           \__ior_new:N #1
10811           \tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
10812           \__ior_open_stream:Nn #1 {#2}
10813         }
10814 ⟨/package⟩
10815   }
10816 \cs_generate_variant:Nn \__ior_open:Nn { No }
10817 \cs_new_protected:Npn \__ior_open_stream:Nn #1#2
10818   {
10819     \tex_global:D \tex_chardef:D #1 = \l__ior_stream_tl \scan_stop:
10820     \prop_gput:NVn \g__ior_streams_prop #1 {#2}
10821     \tex_openin:D #1 #2 \scan_stop:
10822   }
```

(*End definition for* `\__ior_open:Nn` *and* `\__ior_open_stream:Nn`.)

`\ior_close:N`   Closing a stream means getting rid of it at the TeX level and removing from the various
`\ior_close:c`   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.

```
10823 \cs_new_protected:Npn \ior_close:N #1
10824   {
10825     \int_compare:nT { \c_minus_one < #1 < \c_sixteen }
10826       {
10827         \tex_closein:D #1
10828         \prop_gremove:NV \g__ior_streams_prop #1
10829         \seq_if_in:NVF \g__ior_streams_seq #1
10830           { \seq_gpush:NV \g__ior_streams_seq #1 }
10831         \cs_gset_eq:NN #1 \c_term_ior
10832       }
10833   }
10834 \cs_generate_variant:Nn \ior_close:N { c }
```

(*End definition for* `\ior_close:N`. *This function is documented on page 174.*)

`\ior_list_streams:`   Show the property lists, but with some "pretty printing". See the l3msg module. The
`\__ior_list_streams:Nn`   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 (in fact a question mark) otherwise. The code of the
                            message show-streams takes care of translating ior/iow to English. The list of streams
                            is formatted using `\__msg_show_item_unbraced:nn`.

```
10835 \cs_new_protected:Npn \ior_list_streams:
10836   { \__ior_list_streams:Nn \g__ior_streams_prop { ior } }
10837 \cs_new_protected:Npn \__ior_list_streams:Nn #1#2
10838   {
10839     \__msg_show_pre:nnxxxx { LaTeX / kernel } { show-streams }
10840       {#2} { \prop_if_empty:NF #1 { ? } } { } { }
10841     \__msg_show_wrap:n
10842       { \prop_map_function:NN #1 \__msg_show_item_unbraced:nn }
10843   }
```

531

(*End definition for* `\ior_list_streams:` *and* `\__ior_list_streams:Nn`. *These functions are documented on page 174.*)

20.2.3 Reading input

`\if_eof:w` The primitive conditional

```
10844 \cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

(*End definition for* `\if_eof:w`.)

`\ior_if_eof_p:N`
`\ior_if_eof:NTF` To test if some particular input stream is exhausted the following conditional is provided.

```
10845 \prg_new_conditional:Nnn \ior_if_eof:N { p , T , F , TF }
10846   {
10847     \cs_if_exist:NTF #1
10848       {
10849         \if_int_compare:w #1 = \c_sixteen
10850           \prg_return_true:
10851         \else:
10852           \if_eof:w #1
10853             \prg_return_true:
10854           \else:
10855             \prg_return_false:
10856           \fi:
10857         \fi:
10858       }
10859       { \prg_return_true: }
10860   }
```

(*End definition for* `\ior_if_eof:NTF`. *This function is documented on page 175.*)

`\ior_get:NN` And here we read from files.

```
10861 \cs_new_protected:Npn \ior_get:NN #1#2
10862   { \tex_read:D #1 to #2 }
```

(*End definition for* `\ior_get:NN`. *This function is documented on page 174.*)

`\ior_get_str:NN` Reading as strings is a more complicated wrapper, as we wish to remove the endline character.

```
10863 \cs_new_protected:Npn \ior_get_str:NN #1#2
10864   {
10865     \use:x
10866       {
10867         \int_set_eq:NN \tex_endlinechar:D \c_minus_one
10868         \exp_not:n { \etex_readline:D #1 to #2 }
10869         \int_set:Nn \tex_endlinechar:D { \int_use:N \tex_endlinechar:D }
10870       }
10871   }
```

(*End definition for* `\ior_get_str:NN`. *This function is documented on page 175.*)

`\g__file_internal_ior` Needed by the higher-level code, but cannot be created until here.

```
10872 \ior_new:N \g__file_internal_ior
```

(*End definition for* `\g__file_internal_ior`.)

20.3   Output operations

10873 ⟨@@=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.

20.3.1   Variables and constants

\c_log_iow   Here we allocate two output streams for writing to the transcript file only (\c_log_iow)
\c_term_iow   and to both the terminal and transcript file (\c_term_iow).

```
10874 \cs_new_eq:NN \c_log_iow  \c_minus_one
10875 \int_const:Nn \c_term_iow { 128 }
```

(*End definition for* \c_log_iow *and* \c_term_iow. *These variables are documented on page* 178.)

\g__iow_streams_seq   A list of the currently-available output streams to be used as a stack.

```
10876 \seq_new:N \g__iow_streams_seq
10877 ⟨*initex⟩
10878 \seq_set_eq:NN \g__iow_streams_seq \g__ior_streams_seq
10879 \cs_if_exist:NT \luatex_directlua:D
10880   {
10881     \int_compare:nNnT \luatex_luatexversion:D > { 80 }
10882       {
10883         \int_step_inline:nnnn { 16 } { 1 } { 127 }
10884           {
10885             \seq_gput_right:Nn \g__iow_streams_seq {#1}
10886           }
10887       }
10888   }
10889 ⟨/initex⟩
```

(*End definition for* \g__iow_streams_seq.)

\l__iow_stream_tl   Used to recover the raw stream number from the stack.

```
10890 \tl_new:N \l__iow_stream_tl
```

(*End definition for* \l__iow_stream_tl.)

\g__iow_streams_prop   As for reads with the appropriate adjustment of the register numbers to check on.

```
10891 \prop_new:N \g__iow_streams_prop
10892 ⟨*package⟩
10893 \int_step_inline:nnnn
10894   { \c_zero }
10895   { \c_one }
10896   {
10897     \cs_if_exist:NTF \normalend
10898       { \tex_count:D 39 \scan_stop: }
10899       {
10900         \tex_count:D 17 \scan_stop:
10901           \cs_if_exist:NT \loccount { - \c_one }
10902       }
10903   }
10904   {
10905     \prop_gput:Nnn \g__iow_streams_prop {#1} { Reserved~by~format }
10906   }
10907 ⟨/package⟩
```

(*End definition for* \g__iow_streams_prop.)

20.4 Stream management

\iow_new:N Reserving a new stream is done by defining the name as equal to writing to the terminal:
\iow_new:c odd but at least consistent.

```
10908 \cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
10909 \cs_generate_variant:Nn \iow_new:N { c }
```

(*End definition for* \iow_new:N. *This function is documented on page 173.*)

__iow_new:N As for read streams, copy \newwrite in package mode, making sure that it is not \outer.

```
10910 ⟨*package⟩
10911 \exp_args:NNf \cs_new_protected:Npn \__iow_new:N
10912     { \exp_args:NNc \exp_after:wN \exp_stop_f: { newwrite } }
10913 ⟨/package⟩
```

(*End definition for* __iow_new:N.)

\iow_open:Nn The same idea as for reading, but without the path and without the need to allow for a
\iow_open:cn conditional version.
__iow_open:Nn
__iow_open_stream:Nn

```
10914 \cs_new_protected:Npn \iow_open:Nn #1#2
10915     { \__file_name_sanitize:nn {#2} { \__iow_open:Nn #1 } }
10916 \cs_generate_variant:Nn \iow_open:Nn { c }
10917 \cs_new_protected:Npn \__iow_open:Nn #1#2
10918     {
10919         \iow_close:N #1
10920         \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
10921             { \__iow_open_stream:Nn #1 {#2} }
10922 ⟨*initex⟩
10923             { \__msg_kernel_fatal:nn { kernel } { output-streams-exhausted } }
10924 ⟨/initex⟩
10925 ⟨*package⟩
10926             {
10927                 \__iow_new:N #1
10928                 \tl_set:Nx \l__iow_stream_tl { \int_eval:n {#1} }
10929                 \__iow_open_stream:Nn #1 {#2}
10930             }
10931 ⟨/package⟩
10932     }
10933 \cs_generate_variant:Nn \__iow_open:Nn { No }
10934 \cs_new_protected:Npn \__iow_open_stream:Nn #1#2
10935     {
10936         \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:
10937         \prop_gput:NVn \g__iow_streams_prop #1 {#2}
10938         \tex_immediate:D \tex_openout:D #1 #2 \scan_stop:
10939     }
```

(*End definition for* \iow_open:Nn, __iow_open:Nn, *and* __iow_open_stream:Nn. *These functions are documented on page 173.*)

\iow_close:N Closing a stream is not quite the reverse of opening one. First, the close operation is
\iow_close:c 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.

```
10940 \cs_new_protected:Npn \iow_close:N #1
10941     {
10942         \int_compare:nT { \c_minus_one < #1 < \c_sixteen }
```

534

```
10943            {
10944              \tex_immediate:D \tex_closeout:D #1
10945              \prop_gremove:NV \g__iow_streams_prop #1
10946              \seq_if_in:NVF \g__iow_streams_seq #1
10947                { \seq_gpush:NV \g__iow_streams_seq #1 }
10948              \cs_gset_eq:NN #1 \c_term_ior
10949            }
10950      }
10951  \cs_generate_variant:Nn \iow_close:N { c }
```

(*End definition for* \iow_close:N. *This function is documented on page* *174.*)

\iow_list_streams: Done as for input, but with a copy of the auxiliary so the name is correct.
__iow_list_streams:Nn
```
10952  \cs_new_protected:Npn \iow_list_streams:
10953    { \__iow_list_streams:Nn \g__iow_streams_prop { iow } }
10954  \cs_new_eq:NN \__iow_list_streams:Nn \__ior_list_streams:Nn
```

(*End definition for* \iow_list_streams: *and* __iow_list_streams:Nn. *These functions are documented on page* *174.*)

20.4.1   Deferred writing

\iow_shipout_x:Nn First the easy part, this is the primitive, which expects its argument to be braced.
\iow_shipout_x:Nx
\iow_shipout_x:cn
```
10955  \cs_new_protected:Npn \iow_shipout_x:Nn #1#2
10956    { \tex_write:D #1 {#2} }
```
\iow_shipout_x:cx
```
10957  \cs_generate_variant:Nn \iow_shipout_x:Nn { c, Nx, cx }
```

(*End definition for* \iow_shipout_x:Nn. *This function is documented on page* *176.*)

\iow_shipout:Nn With ε-TEX available deferred writing without expansion is easy.
\iow_shipout:Nx
\iow_shipout:cn
```
10958  \cs_new_protected:Npn \iow_shipout:Nn #1#2
10959    { \tex_write:D #1 { \exp_not:n {#2} } }
```
\iow_shipout:cx
```
10960  \cs_generate_variant:Nn \iow_shipout:Nn { c, Nx, cx }
```

(*End definition for* \iow_shipout:Nn. *This function is documented on page* *175.*)

20.4.2   Immediate writing

__iow_with:Nnn If the integer #1 is equal to #2, just leave #3 in the input stream. Otherwise, pass the old
__iow_with_aux:nNnn value to an auxiliary, which sets the integer to the new value, runs the code, and restores
the integer.
```
10961  \cs_new_protected:Npn \__iow_with:Nnn #1#2
10962    {
10963      \int_compare:nNnTF {#1} = {#2}
10964        { \use:n }
10965        { \exp_args:No \__iow_with_aux:nNnn { \int_use:N #1 } #1 {#2} }
10966    }
10967  \cs_new_protected:Npn \__iow_with_aux:nNnn #1#2#3#4
10968    {
10969      \int_set:Nn #2 {#3}
10970      #4
10971      \int_set:Nn #2 {#1}
10972    }
```

(*End definition for* __iow_with:Nnn *and* __iow_with_aux:nNnn.)

`\iow_now:Nn`
`\iow_now:Nx`
`\iow_now:cn`
`\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 `\__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.

```
10973 \cs_new_protected:Npn \iow_now:Nn #1#2
10974   {
10975     \__iow_with:Nnn \tex_newlinechar:D { '\^^J }
10976       { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
10977   }
10978 \cs_generate_variant:Nn \iow_now:Nn { c, Nx, cx }
```

(*End definition for* `\iow_now:Nn`. *This function is documented on page 175.*)

`\iow_log:n`
`\iow_log:x`
`\iow_term:n`
`\iow_term:x`

Writing to the log and the terminal directly are relatively easy.

```
10979 \cs_set_protected:Npn \iow_log:x  { \iow_now:Nx \c_log_iow  }
10980 \cs_new_protected:Npn \iow_log:n  { \iow_now:Nn \c_log_iow  }
10981 \cs_set_protected:Npn \iow_term:x { \iow_now:Nx \c_term_iow }
10982 \cs_new_protected:Npn \iow_term:n { \iow_now:Nn \c_term_iow }
```

(*End definition for* `\iow_log:n` *and* `\iow_term:n`. *These functions are documented on page 175.*)

20.4.3 Special characters for writing

`\iow_newline:`

Global variable holding the character that forces a new line when something is written to an output stream.

```
10983 \cs_new:Npn \iow_newline: { ^^J }
```

(*End definition for* `\iow_newline:`. *This function is documented on page 176.*)

`\iow_char:N`

Function to write any escaped char to an output stream.

```
10984 \cs_new_eq:NN \iow_char:N \cs_to_str:N
```

(*End definition for* `\iow_char:N`. *This function is documented on page 176.*)

20.4.4 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 TeXLive and MikTeX.

```
10985 \int_new:N  \l_iow_line_count_int
10986 \int_set:Nn \l_iow_line_count_int { 78 }
```

(*End definition for* `\l_iow_line_count_int`. *This variable is documented on page 177.*)

`\l__iow_target_count_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.

```
10987 \int_new:N \l__iow_target_count_int
```

(*End definition for* \l__iow_target_count_int.)

\l__iow_current_line_int
\l__iow_current_word_int
\l__iow_current_indentation_int

These store the number of characters in the line and word currently being constructed, and the current indentation, respectively.

```
10988 \int_new:N \l__iow_current_line_int
10989 \int_new:N \l__iow_current_word_int
10990 \int_new:N \l__iow_current_indentation_int
```

(*End definition for* \l__iow_current_line_int, \l__iow_current_word_int, *and* \l__iow_current_-
indentation_int.)

\l__iow_current_line_tl
\l__iow_current_word_tl
\l__iow_current_indentation_tl

These hold the current line of text and current word, and a number of spaces for indentation, respectively.

```
10991 \tl_new:N \l__iow_current_line_tl
10992 \tl_new:N \l__iow_current_word_tl
10993 \tl_new:N \l__iow_current_indentation_tl
```

(*End definition for* \l__iow_current_line_tl, \l__iow_current_word_tl, *and* \l__iow_current_-
indentation_tl.)

\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.

```
10994 \tl_new:N \l__iow_wrap_tl
```

(*End definition for* \l__iow_wrap_tl.)

\l__iow_newline_tl

The token list inserted to produce the new line, with the ⟨*run-on text*⟩.

```
10995 \tl_new:N \l__iow_newline_tl
```

(*End definition for* \l__iow_newline_tl.)

\l__iow_line_start_bool

Boolean to avoid adding a space at the beginning of forced newlines, and to know when to add the indentation.

```
10996 \bool_new:N \l__iow_line_start_bool
```

(*End definition for* \l__iow_line_start_bool.)

\c_catcode_other_space_tl

Create a space with category code 12: an "other" space.

```
10997 \tl_const:Nx \c_catcode_other_space_tl { \char_generate:nn { '\ } { 12 } }
```

(*End definition for* \c_catcode_other_space_tl. *This function is documented on page* 177.)

\c__iow_wrap_marker_tl
\c__iow_wrap_end_marker_tl
\c__iow_wrap_newline_marker_tl
\c__iow_wrap_indent_marker_tl
\c__iow_wrap_unindent_marker_tl

Every special action of the wrapping code is preceded by 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 nicer.

```
10998 \group_begin:
10999   \int_set_eq:NN \tex_escapechar:D \c_minus_one
11000   \tl_const:Nx \c__iow_wrap_marker_tl
11001     { \tl_to_str:n { \^^I \^^O \^^W \^^_ \^^W \^^R \^^A \^^P } }
11002 \group_end:
11003 \tl_map_inline:nn
11004   { { end } { newline } { indent } { unindent } }
11005   {
11006     \tl_const:cx { c__iow_wrap_ #1 _marker_tl }
```

537

```
11007        {
11008          \c_catcode_other_space_tl
11009          \c__iow_wrap_marker_tl
11010          \c_catcode_other_space_tl
11011          #1
11012          \c_catcode_other_space_tl
11013        }
11014    }
```

(*End definition for* `\c__iow_wrap_marker_tl` *and others.*)

\iow_indent:n
__iow_indent:n
__iow_indent_error:n

We give a (protected) error definition to \iow_indent:n when outside messages. Within wrapped message, it places the instruction for increasing the indentation before its argument, and the instruction for unindenting afterwards. Note that there will be no forced line-break, so the indentation only changes when the next line is started.

```
11015  \cs_new:Npx \__iow_indent:n #1
11016    {
11017      \c__iow_wrap_indent_marker_tl
11018      #1
11019      \c__iow_wrap_unindent_marker_tl
11020    }
11021  \cs_new:Npn \__iow_indent_error:n #1
11022    {
11023      \__msg_kernel_expandable_error:nn { kernel } { indent-outside-wrapping-code }
11024      #1
11025    }
11026  \cs_new_protected:Npn \iow_indent:n { \__iow_indent_error:n }
```

(*End definition for* `\iow_indent:n`, `\__iow_indent:n`, *and* `\__iow_indent_error:n`. *These functions are documented on page 177.*)

\iow_wrap:nnnN
__iow_wrap_set:Nx

The main wrapping function works as follows. First give \\, \␣ and other formatting commands the correct definition for messages, before fully-expanding the input. In package mode, the expansion uses LATEX 2ε's \protect mechanism. Afterwards, set the newline marker (two assignments to fully expand, then convert to a string) and its length, and initialize some registers. There is then a loop over each word in the input, which will do the actual wrapping. After the loop, the resulting text is passed on to the function which has been given as a post-processor. The argument #4 is available for additional set up steps for the output. The definition of \\ and \␣ use 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. The \tl_to_str:N step converts the "other" space back to a normal space.

```
11027  \cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
11028    {
11029      \group_begin:
11030        \int_set_eq:NN \tex_escapechar:D \c_minus_one
11031        \cs_set:Npx \{ { \token_to_str:N \{ }
11032        \cs_set:Npx \# { \token_to_str:N \# }
11033        \cs_set:Npx \} { \token_to_str:N \} }
11034        \cs_set:Npx \% { \token_to_str:N \% }
11035        \cs_set:Npx \~ { \token_to_str:N \~ }
11036        \int_set:Nn \tex_escapechar:D { 92 }
11037        \cs_set_eq:NN \\ \c__iow_wrap_newline_marker_tl
```

538

```
11038            \cs_set_eq:NN \  \c_catcode_other_space_tl
11039            \cs_set_eq:NN \iow_indent:n \__iow_indent:n
11040         #3
11041  ⟨*initex⟩
11042            \tl_set:Nx \l__iow_wrap_tl {#1}
11043  ⟨/initex⟩
11044  ⟨*package⟩
11045            \__iow_wrap_set:Nx \l__iow_wrap_tl {#1}
11046  ⟨/package⟩
```

To warn users that `\iow_indent:n` only works in the first argument of `\iow_wrap:nnnN` reset `\iow_indent:n` to its error definition. Then store a newline character and the run-on text as a string in `\l__iow_newline_tl`, and set some variables. The first line's target count is equal to the length of the whole line. The value `\l__iow_target_count_int` is altered later on by `\__iow_wrap_set_target:`.

```
11047            \cs_set_eq:NN \iow_indent:n \__iow_indent_error:n
11048            \tl_set:Nx \l__iow_newline_tl { \iow_newline: #2 }
11049            \tl_set:Nx \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
11050            \int_set_eq:NN \l__iow_target_count_int \l_iow_line_count_int
11051            \tl_clear:N \l__iow_current_indentation_tl
11052            \int_zero:N \l__iow_current_line_int
11053            \tl_set:Nn \l__iow_current_line_tl { \use_none:n }
11054            \bool_set_true:N \l__iow_line_start_bool
```

After some setup above (in particular the odd setting of the current line to `\use_none:n`), a loop goes through space-delimited words in the message, recognizing special markers. To make sure that the first line behaves identically to others, start with a newline marker: the `\use_none:n` above avoids actually getting a new line in the output.

```
11055            \use:x
11056              {
11057                \exp_not:n { \tl_clear:N \l__iow_wrap_tl }
11058                \__iow_wrap_loop:w
11059                \tl_to_str:N \c__iow_wrap_newline_marker_tl
11060                \tl_to_str:N \l__iow_wrap_tl
11061                \tl_to_str:N \c__iow_wrap_end_marker_tl
11062                \c_space_tl \c_space_tl
11063                \exp_not:N \q_stop
11064              }
11065        \exp_args:NNo \group_end:
11066        #4 \l__iow_wrap_tl
11067    }
```

As using the generic loader will mean that `\protected@edef` is not available, it's not placed directly in the wrap function but is set up as an auxiliary. In the generic loader this can then be redefined.

```
11068  ⟨*package⟩
11069  \cs_new_eq:NN \__iow_wrap_set:Nx \protected@edef
11070  ⟨/package⟩
```

(*End definition for* `\iow_wrap:nnnN` *and* `\__iow_wrap_set:Nx`*. These functions are documented on page* [177](#)*.*)

`\__iow_wrap_set_target:` This is called at the beginning of every line (both those forced by `\\` and those due to line-breaking). The initial call does nothing except redefine `\__iow_wrap_set_target:` itself (within the group in which `\iow_wrap:nnnN` works). The next call (at the beginning

of the second line) disables any later call and sets the `\l__iow_target_count_int` to the correct value, namely the `\l_iow_line_count_int` shortened by the length of the run-on text (the shift by 1 is due to the presence of `\iow_newline:` in `\l__iow_newline_tl`). This is a bit of a hack to measure the string length of the run on text without the l3str module (which is still experimental). This should be replaced once the string module is finalised with something a little cleaner.

```
11071 \cs_new_protected:Npn \__iow_wrap_set_target:
11072   {
11073     \cs_set_protected:Npn \__iow_wrap_set_target:
11074       {
11075         \cs_set_protected:Npn \__iow_wrap_set_target: { }
11076         \tl_replace_all:Nnn \l__iow_newline_tl { ~ } { \c_space_tl }
11077         \int_set:Nn \l__iow_target_count_int
11078           { \l_iow_line_count_int - \tl_count:N \l__iow_newline_tl + \c_one }
11079       }
11080   }
```

(*End definition for* `\__iow_wrap_set_target:`.)

`\__iow_wrap_loop:w`   The loop grabs one word in the input, and checks whether it is the special marker, or a normal word.

```
11081 \cs_new_protected:Npn \__iow_wrap_loop:w #1 ~ %
11082   {
11083     \tl_set:Nn \l__iow_current_word_tl {#1}
11084     \tl_if_eq:NNTF \l__iow_current_word_tl \c__iow_wrap_marker_tl
11085       { \__iow_wrap_special:w }
11086       { \__iow_wrap_word: }
11087   }
```

(*End definition for* `\__iow_wrap_loop:w`.)

`\__iow_wrap_word:`
`\__iow_wrap_word_fits:`
`\__iow_wrap_word_newline:`
For a normal word, update the line count, then test if the current word would fit in the current line, and call the appropriate function. If the word fits in the current line, add it to the line, preceded by a space unless it is the first word of the line. Otherwise, the current line is added to the result, with the run-on text. The current word (and its character count) are then put in the new line.

```
11088 \cs_new_protected:Npn \__iow_wrap_word:
11089   {
11090     \int_set:Nn \l__iow_current_word_int
11091       { \exp_args:No  \str_count_ignore_spaces:n \l__iow_current_word_tl }
11092     \int_add:Nn \l__iow_current_line_int { \l__iow_current_word_int }
11093     \int_compare:nNnTF \l__iow_current_line_int < \l__iow_target_count_int
11094       { \__iow_wrap_word_fits: }
11095       { \__iow_wrap_word_newline: }
11096     \__iow_wrap_loop:w
11097   }
11098 \cs_new_protected:Npn \__iow_wrap_word_fits:
11099   {
11100     \bool_if:NTF \l__iow_line_start_bool
11101       {
11102         \bool_set_false:N \l__iow_line_start_bool
11103         \tl_put_right:Nx \l__iow_current_line_tl
11104           { \l__iow_current_indentation_tl \l__iow_current_word_tl }
```

```
11105        \int_add:Nn \l__iow_current_line_int
11106          { \l__iow_current_indentation_int }
11107        }
11108        {
11109          \tl_put_right:Nx \l__iow_current_line_tl
11110            { ~ \l__iow_current_word_tl }
11111          \int_incr:N \l__iow_current_line_int
11112        }
11113    }
11114 \cs_new_protected:Npn \__iow_wrap_word_newline:
11115    {
11116      \__iow_wrap_set_target:
11117      \tl_put_right:Nx \l__iow_wrap_tl
11118        { \l__iow_current_line_tl \l__iow_newline_tl }
11119      \int_set:Nn \l__iow_current_line_int
11120        {
11121          \l__iow_current_word_int
11122          + \l__iow_current_indentation_int
11123        }
11124      \tl_set:Nx \l__iow_current_line_tl
11125        { \l__iow_current_indentation_tl \l__iow_current_word_tl }
11126    }
```

(*End definition for* __iow_wrap_word: , __iow_wrap_word_fits: , *and* __iow_wrap_word_newline:.)

__iow_wrap_special:w
__iow_wrap_newline:w
__iow_wrap_indent:w
__iow_wrap_unindent:w
__iow_wrap_end:w

When the "special" marker is encountered, read what operation to perform, as a space-delimited argument, perform it, and remember to loop. In fact, to avoid spurious spaces when two special actions follow each other, we look ahead for another copy of the marker. Forced newlines are almost identical to those caused by overflow, except that here the word is empty. To indent more, add four spaces to the start of the indentation token list. To reduce indentation, rebuild the indentation token list using \prg_replicate:nn. At the end, we simply save the last line (without the run-on text), and prevent the loop.

```
11127 \cs_new_protected:Npn \__iow_wrap_special:w #1 ~ #2 ~ #3 ~ %
11128    {
11129      \use:c { __iow_wrap_#1: }
11130      \str_if_eq_x:nnTF { #2~#3 } { ~ \c__iow_wrap_marker_tl }
11131        { \__iow_wrap_special:w }
11132        { \__iow_wrap_loop:w #2 ~ #3 ~ }
11133    }
11134 \cs_new_protected:Npn \__iow_wrap_newline:
11135    {
11136      \__iow_wrap_set_target:
11137      \tl_put_right:Nx \l__iow_wrap_tl
11138        { \l__iow_current_line_tl \l__iow_newline_tl }
11139      \int_zero:N \l__iow_current_line_int
11140      \tl_clear:N \l__iow_current_line_tl
11141      \bool_set_true:N \l__iow_line_start_bool
11142    }
11143 \cs_new_protected:Npx \__iow_wrap_indent:
11144    {
11145      \int_add:Nn \l__iow_current_indentation_int \c_four
11146      \tl_put_right:Nx \exp_not:N \l__iow_current_indentation_tl
11147        { \c_space_tl \c_space_tl \c_space_tl \c_space_tl }
11148    }
```

541

```
11149 \cs_new_protected:Npn \__iow_wrap_unindent:
11150   {
11151     \int_sub:Nn \l__iow_current_indentation_int \c_four
11152     \tl_set:Nx \l__iow_current_indentation_tl
11153       { \prg_replicate:nn \l__iow_current_indentation_int { ~ } }
11154   }
11155 \cs_new_protected:Npn \__iow_wrap_end:
11156   {
11157     \tl_put_right:Nx \l__iow_wrap_tl
11158       { \l__iow_current_line_tl }
11159     \use_none_delimit_by_q_stop:w
11160   }
```

(*End definition for* `\__iow_wrap_special:w` *and others.*)

20.5 Messages

```
11161 \__msg_kernel_new:nnnn { kernel } { file-not-found }
11162   { File~'#1'~not~found. }
11163   {
11164     The~requested~file~could~not~be~found~in~the~current~directory,~
11165     in~the~TeX~search~path~or~in~the~LaTeX~search~path.
11166   }
11167 \__msg_kernel_new:nnnn { kernel } { input-streams-exhausted }
11168   { Input~streams~exhausted }
11169   {
11170     TeX~can~only~open~up~to~16~input~streams~at~one~time.\\
11171     All~16~are~currently~in~use,~and~something~wanted~to~open~
11172     another~one.
11173   }
11174 \__msg_kernel_new:nnnn { kernel } { output-streams-exhausted }
11175   { Output~streams~exhausted }
11176   {
11177     TeX~can~only~open~up~to~16~output~streams~at~one~time.\\
11178     All~16~are~currently~in~use,~and~something~wanted~to~open~
11179     another~one.
11180   }
11181 \__msg_kernel_new:nnnn { kernel } { unbalanced-quote-in-filename }
11182   { Unbalanced~quotes~in~file~name~'#1'. }
11183   {
11184     File~names~must~contain~balanced~numbers~of~quotes~(").
11185   }
11186 \__msg_kernel_new:nnn { kernel } { indent-outside-wrapping-code }
11187   { Only~\iow_wrap:nnnN~(arg~1)~allows~\iow_indent:n }
11188 ⟨/initex | package⟩
```

21   l3fp implementation

Nothing to see here: everything is in the subfiles!

22   l3fp-aux implementation

```
11189 ⟨*initex | package⟩
```

22.1   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 will be used in expressions, and in this context will be 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.

Since floating point numbers are always accessed by the various operations using f-expansion, we can safely let them be protected: x-expansion will then leave them untouched. However, when used directly without an accessor function, floating points should produce an error. \s__fp will do nothing, and __fp_chk:w produces an error.

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. The various possibilities will be distinguished by their ⟨*case*⟩, which is a single digit:[9]

0 zeros: +0 and -0,

1 "normal" numbers (positive and negative),

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, at most \c__fp_max_exponent_int = 10000 in absolute value. The body consists in four blocks of exactly 4 digits, $0000 \leq ⟨X_i⟩ \leq 9999$, such that

$$⟨X⟩ = (-1)^{⟨sign⟩} 10^{-⟨exponent⟩} \sum_{i=1}^{4} ⟨X_i⟩ 10^{-4i}$$

and such that the ⟨*exponent*⟩ is minimal. This implies $1000 \leq ⟨X_1⟩ \leq 9999$.

---

[9]Bruno: I need to implement subnormal numbers. Also, quiet and signalling nan must be better distinguished.

Table 1: Internal representation of floating point numbers.

Representation	Meaning
0 0 \s__fp_... ;	Positive zero.
0 2 \s__fp_... ;	Negative zero.
1 0 {⟨exponent⟩} {⟨X₁⟩} {⟨X₂⟩} {⟨X₃⟩} {⟨X₄⟩} ;	Positive floating point.
1 2 {⟨exponent⟩} {⟨X₁⟩} {⟨X₂⟩} {⟨X₃⟩} {⟨X₄⟩} ;	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.

22.2 Internal storage of floating points numbers

A floating point number $\langle X \rangle$ is stored as

$\s__fp \ \__fp_chk:w$ ⟨*case*⟩ ⟨*sign*⟩ ⟨*body*⟩ ;

Here, ⟨*case*⟩ is 0 for ± 0, 1 for normal numbers, 2 for $\pm\infty$, and 3 for `nan`, and ⟨*sign*⟩ is 0 for positive numbers, 1 for `nan`s, and 2 for negative numbers. The ⟨*body*⟩ of normal numbers is $\{\langle exponent \rangle\} \ \{\langle X_1 \rangle\} \ \{\langle X_2 \rangle\} \ \{\langle X_3 \rangle\} \ \{\langle X_4 \rangle\}$, with

$$\langle X \rangle = (-1)^{\langle sign \rangle} 10^{-\langle exponent \rangle} \sum_i \langle X_i \rangle 10^{-4i}.$$

Calculations are done in base 10000, *i.e.* one myriad. The ⟨*exponent*⟩ lies between $\pm \c__fp_max_exponent_int = \pm 10000$ inclusive.

Additionally, positive and negative floating point numbers may only be stored with $1000 \leq \langle X_1 \rangle < 10000$. This requirement is necessary in order to preserve accuracy and speed.

22.3 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.

```
11191 \cs_new:Npn \__fp_use_none_stop_f:n #1 { \exp_stop_f: }
```

(*End definition for* __fp_use_none_stop_f:n.)

__fp_use_s:n   Those functions place a semicolon after one or two arguments (typically digits).
__fp_use_s:nn

```
11192 \cs_new:Npn \__fp_use_s:n #1 { #1; }
11193 \cs_new:Npn \__fp_use_s:nn #1#2 { #1#2; }
```

(*End definition for* __fp_use_s:n *and* __fp_use_s:nn.)

__fp_use_none_until_s:w   Those functions select specific arguments among a set of arguments delimited by a semi-
__fp_use_i_until_s:nw   colon.
__fp_use_ii_until_s:nnw

```
11194 \cs_new:Npn \__fp_use_none_until_s:w #1; { }
11195 \cs_new:Npn \__fp_use_i_until_s:nw #1#2; {#1}
11196 \cs_new:Npn \__fp_use_ii_until_s:nnw #1#2#3; {#2}
```

(*End 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.

```
11197 \cs_new:Npn \__fp_reverse_args:Nww #1 #2; #3; { #1 #3; #2; }
```

(*End 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.

```
11198 \cs_new:Npn \__fp_rrot:www #1; #2; #3; { #2; #3; #1; }
```

(*End definition for* __fp_rrot:www.)

__fp_use_i:ww   Many internal functions take arguments delimited by semicolons, and it is occasionally
__fp_use_i:www   useful to remove one or two such arguments.

```
11199 \cs_new:Npn \__fp_use_i:ww #1; #2; { #1; }
11200 \cs_new:Npn \__fp_use_i:www #1; #2; #3; { #1; }
```

(*End definition for* __fp_use_i:ww *and* __fp_use_i:www.)

22.4   Constants, and structure of floating points

\s__fp   Floating points numbers all start with \s__fp __fp_chk:w, where \s__fp is equal to
__fp_chk:w   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 x-expansion. However, when typeset, \s__fp does nothing, and __fp_chk:w is expanded. We define __fp_chk:w to produce an error.

```
11201 \__scan_new:N \s__fp
11202 \cs_new_protected:Npn \__fp_chk:w #1 ;
11203   {
11204     \__msg_kernel_error:nnx { kernel } { misused-fp }
11205       { \fp_to_tl:n { \s__fp \__fp_chk:w #1 ; } }
11206   }
```

(*End definition for* \s__fp *and* __fp_chk:w.)

\s__fp_mark   Aliases of \tex_relax:D, used to terminate expressions.
\s__fp_stop

```
11207 \__scan_new:N \s__fp_mark
11208 \__scan_new:N \s__fp_stop
```

(*End definition for* \s__fp_mark *and* \s__fp_stop.)

\s__fp_invalid   A couple of scan marks used to indicate where special floating point numbers come from.
\s__fp_underflow
\s__fp_overflow
\s__fp_division
\s__fp_exact

```
11209 \__scan_new:N \s__fp_invalid
11210 \__scan_new:N \s__fp_underflow
11211 \__scan_new:N \s__fp_overflow
11212 \__scan_new:N \s__fp_division
11213 \__scan_new:N \s__fp_exact
```

(*End definition for* \s__fp_invalid *and others.*)

\c_zero_fp   The special floating points. All of them have the form
\c_minus_zero_fp
\c_inf_fp   \s__fp __fp_chk:w ⟨*case*⟩ ⟨*sign*⟩ \s__fp_... ;
\c_minus_inf_fp
\c_nan_fp

where the dots in `\s__fp_...` are one of `invalid`, `underflow`, `overflow`, `division`, `exact`, describing how the floating point was created. We define the floating points here as "exact".

```
11214 \tl_const:Nn \c_zero_fp       { \s__fp \__fp_chk:w 0 0 \s__fp_exact ; }
11215 \tl_const:Nn \c_minus_zero_fp { \s__fp \__fp_chk:w 0 2 \s__fp_exact ; }
11216 \tl_const:Nn \c_inf_fp        { \s__fp \__fp_chk:w 2 0 \s__fp_exact ; }
11217 \tl_const:Nn \c_minus_inf_fp  { \s__fp \__fp_chk:w 2 2 \s__fp_exact ; }
11218 \tl_const:Nn \c_nan_fp        { \s__fp \__fp_chk:w 3 1 \s__fp_exact ; }
```

(*End definition for* `\c_zero_fp` *and others. These variables are documented on page 186.*)

`\c__fp_max_exponent_int`   Normal floating point numbers have an exponent at most `max_exponent` in absolute value. Larger numbers are rounded to $\pm\infty$. Smaller numbers are subnormal (not implemented yet), and digits beyond $10^{-\texttt{max\_exponent}}$ are rounded away, hence the true minimum exponent is $-\texttt{max\_exponent} - 16$; beyond this, numbers are rounded to zero. Why this choice of limits? When computing $(a \cdot 10^n)(b \cdot 10^p)$, we need to evaluate $\log(a \cdot 10^n) = \log(a) + n\log(10)$ as a fixed point number, which we manipulate as blocks of 4 digits. Multiplying such a fixed point number by $n < 10000$ is much cheaper than larger n, because we can multiply n with each block safely.

```
11219 \int_const:Nn \c__fp_max_exponent_int { 10000 }
```

(*End definition for* `\c__fp_max_exponent_int`.)

`\__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.

```
11220 \cs_new:Npn \__fp_zero_fp:N #1
11221   { \s__fp \__fp_chk:w 0 #1 \s__fp_underflow ; }
11222 \cs_new:Npn \__fp_inf_fp:N #1
11223   { \s__fp \__fp_chk:w 2 #1 \s__fp_overflow ; }
```

(*End definition for* `\__fp_zero_fp:N` *and* `\__fp_inf_fp:N`.)

`\__fp_max_fp:N`
`\__fp_min_fp:N`   In some cases, we need to output the smallest or biggest positive or negative finite numbers.

```
11224 \cs_new:Npn \__fp_min_fp:N #1
11225   {
11226     \s__fp \__fp_chk:w 1 #1
11227       { \int_eval:n { - \c__fp_max_exponent_int } }
11228       {1000} {0000} {0000} {0000} ;
11229   }
11230 \cs_new:Npn \__fp_max_fp:N #1
11231   {
11232     \s__fp \__fp_chk:w 1 #1
11233       { \int_use:N \c__fp_max_exponent_int }
11234       {9999} {9999} {9999} {9999} ;
11235   }
```

(*End definition for* `\__fp_max_fp:N` *and* `\__fp_min_fp:N`.)

`\__fp_exponent:w`   For normal numbers, the function expands to the exponent, otherwise to 0.

```
11236 \cs_new:Npn \__fp_exponent:w \s__fp \__fp_chk:w #1
11237   {
11238     \if_meaning:w 1 #1
11239       \exp_after:wN \__fp_use_ii_until_s:nnw
11240     \else:
```

```
11241                \exp_after:wN \__fp_use_i_until_s:nw
11242                \exp_after:wN 0
11243              \fi:
11244          }
```

(*End 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.

```
11245  \cs_new:Npn \__fp_neg_sign:N #1
11246    { \__int_eval:w \c_two - #1 \__int_eval_end: }
```

(*End definition for* `\__fp_neg_sign:N`.)

22.5 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.

```
11247  \cs_new:Npn \__fp_sanitize:Nw #1 #2;
11248    {
11249      \if_case:w
11250        \if_int_compare:w #2 > \c__fp_max_exponent_int \c_one \else:
11251        \if_int_compare:w #2 < - \c__fp_max_exponent_int \c_two \else:
11252        \if_meaning:w 1 #1 \c_three \else: \c_zero \fi: \fi: \fi:
11253      \or: \exp_after:wN \__fp_overflow:w
11254      \or: \exp_after:wN \__fp_underflow:w
11255      \or: \exp_after:wN \__fp_sanitize_zero:w
11256      \fi:
11257      \s__fp \__fp_chk:w 1 #1 {#2}
11258    }
11259  \cs_new:Npn \__fp_sanitize:wN #1; #2 { \__fp_sanitize:Nw #2 #1; }
11260  \cs_new:Npn \__fp_sanitize_zero:w \s__fp \__fp_chk:w #1 #2 #3;
11261    { \c_zero_fp }
```

(*End definition for* `\__fp_sanitize:Nw`, `\__fp_sanitize:wN`, *and* `\__fp_sanitize_zero:w`.)

22.6 Expanding after a floating point number

`\__fp_exp_after_o:w`
`\__fp_exp_after_o:nw`
`\__fp_exp_after_f:nw`

Places ⟨*tokens*⟩ (empty in the case of `\__fp_exp_after_o:w`) between the ⟨*floating point*⟩ and the ⟨*more tokens*⟩, then hits those tokens with either o-expansion (one `\exp_-after:wN`) 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.

```
11262  \cs_new:Npn \__fp_exp_after_o:w \s__fp \__fp_chk:w #1
11263    {
11264      \if_meaning:w 1 #1
11265        \exp_after:wN \__fp_exp_after_normal:nNNw
11266      \else:
11267        \exp_after:wN \__fp_exp_after_special:nNNw
11268      \fi:
```

```
11269        { }
11270        #1
11271      }
11272  \cs_new:Npn \__fp_exp_after_o:nw #1 \s__fp \__fp_chk:w #2
11273      {
11274        \if_meaning:w 1 #2
11275          \exp_after:wN \__fp_exp_after_normal:nNNw
11276        \else:
11277          \exp_after:wN \__fp_exp_after_special:nNNw
11278        \fi:
11279        { #1 }
11280        #2
11281      }
11282  \cs_new:Npn \__fp_exp_after_f:nw #1 \s__fp \__fp_chk:w #2
11283      {
11284        \if_meaning:w 1 #2
11285          \exp_after:wN \__fp_exp_after_normal:nNNw
11286        \else:
11287          \exp_after:wN \__fp_exp_after_special:nNNw
11288        \fi:
11289        { \exp:w \exp_end_continue_f:w #1 }
11290        #2
11291      }
```

(*End definition for* __fp_exp_after_o:w, __fp_exp_after_o:nw, *and* __fp_exp_after_f:nw.)

__fp_exp_after_special:nNNw   Special floating point numbers are easy to jump over since they contain few tokens.

```
11292  \cs_new:Npn \__fp_exp_after_special:nNNw #1#2#3#4;
11293      {
11294        \exp_after:wN \s__fp
11295        \exp_after:wN \__fp_chk:w
11296        \exp_after:wN #2
11297        \exp_after:wN #3
11298        \exp_after:wN #4
11299        \exp_after:wN ;
11300        #1
11301      }
```

(*End 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.

```
11302  \cs_new:Npn \__fp_exp_after_normal:nNNw #1 1 #2 #3 #4#5#6#7;
11303      {
11304        \exp_after:wN \__fp_exp_after_normal:Nwwww
11305        \exp_after:wN #2
11306        \__int_value:w #3   \exp_after:wN ;
11307        \__int_value:w 1 #4 \exp_after:wN ;
11308        \__int_value:w 1 #5 \exp_after:wN ;
11309        \__int_value:w 1 #6 \exp_after:wN ;
11310        \__int_value:w 1 #7 \exp_after:wN ; #1
11311      }
11312  \cs_new:Npn \__fp_exp_after_normal:Nwwww
```

`#1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;`
11314   `{ \s__fp \__fp_chk:w 1 #1 {#2} {#3} {#4} {#5} {#6} ; }`

(*End definition for* `\__fp_exp_after_normal:nNNw.`)

`\__fp_exp_after_array_f:w`
`\__fp_exp_after_stop_f:nw`

11315 `\cs_new:Npn \__fp_exp_after_array_f:w #1`
11316   `{`
11317   `\cs:w __fp_exp_after \__fp_type_from_scan:N #1 _f:nw \cs_end:`
11318   `{ \__fp_exp_after_array_f:w }`
11319   `#1`
11320   `}`
11321 `\cs_new_eq:NN \__fp_exp_after_stop_f:nw \use_none:nn`

(*End definition for* `\__fp_exp_after_array_f:w` *and* `\__fp_exp_after_stop_f:nw.`)

22.7 Packing digits

When a positive integer `#1` is known to be less than 10^8, the following trick will split 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
  \__int_value:w \__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
\__int_value:w \__int_eval:w - 5 0000
  \exp_after:wN \pack:NNNNNw
  \__int_value:w \__int_eval:w 4 9995 0000
    + 12345 * 6677
  \exp_after:wN \pack:NNNNNw
  \__int_value:w \__int_eval:w 5 0000 0000
    + 12345 * 8899 ;
```

The `\exp_after:wN` triggers `\__int_value:w \__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 \__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 will also work 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

549

range, the result of this second computation will have 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.

```
11322 \int_const:Nn \c__fp_leading_shift_int  { - 5 0000 }
11323 \int_const:Nn \c__fp_middle_shift_int   { 5 0000 *  9999 }
11324 \int_const:Nn \c__fp_trailing_shift_int { 5 0000 * 10000 }
11325 \cs_new:Npn \__fp_pack:NNNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }
```

(*End 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.

```
11326 \int_const:Nn \c__fp_big_leading_shift_int  { - 15 2374 }
11327 \int_const:Nn \c__fp_big_middle_shift_int   { 15 2374 *  9999 }
11328 \int_const:Nn \c__fp_big_trailing_shift_int { 15 2374 * 10000 }
11329 \cs_new:Npn \__fp_pack_big:NNNNNNw #1#2 #3#4#5#6 #7;
11330   { + #1#2#3#4#5#6 ; {#7} }
```

(*End 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.

```
11331 \int_const:Nn \c__fp_Bigg_leading_shift_int  { - 20 0000 }
11332 \int_const:Nn \c__fp_Bigg_middle_shift_int   { 20 0000 *  9999 }
11333 \int_const:Nn \c__fp_Bigg_trailing_shift_int { 20 0000 * 10000 }
11334 \cs_new:Npn \__fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7;
11335   { + #1#2#3#4#5#6 ; {#7} }
```

(*End definition for* `\__fp_pack_Bigg:NNNNNNw` *and others.*)

`\__fp_pack_twice_four:wNNNNNNNN`

Grabs two sets of 4 digits and places them before the semi-colon delimiter. Putting several copies of this function before a semicolon will pack more digits since each will take the digits packed by the others in its first argument.

```
11336 \cs_new:Npn \__fp_pack_twice_four:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
11337   { #1 {#2#3#4#5} {#6#7#8#9} ; }
```

(*End definition for* `\__fp_pack_twice_four:wNNNNNNNN.`)

`\__fp_pack_eight:wNNNNNNNN`

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 will pack more digits since each will take the digits packed by the others in its first argument.

```
11338 \cs_new:Npn \__fp_pack_eight:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
11339   { #1 {#2#3#4#5#6#7#8#9} ; }
```

(*End definition for* `\__fp_pack_eight:wNNNNNNNN.`)

22.8 Decimate (dividing by a power of 10)

`\__fp_decimate:nNnnnn`

Each $\langle X_i \rangle$ consists in 4 digits exactly, and $1000 \leq \langle X_1 \rangle < 9999$. The first argument determines by how much we shift the digits. $\langle f_1 \rangle$ is called as follows: where $0 \leq \langle X'_i \rangle < 10^8 - 1$ are 8 digit numbers, forming the truncation of our number. In other words,

$$\left(\sum_{i=1}^{4} \langle X_i \rangle \cdot 10^{-4i} \cdot 10^{-\langle shift \rangle} - \langle X'_1 \rangle \cdot 10^{-8} + \langle X'_2 \rangle \cdot 10^{-16} \right) \in [0, 10^{-16}).$$

To round properly later, we need to remember some information about the difference. The $\langle rounding \rangle$ 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, $\langle rounding \rangle$ is 1 (not 0), and $\langle X'_1 \rangle \langle X'_2 \rangle$ are both zero.

If the shift is 1, the $\langle rounding \rangle$ 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 $\langle rounding \rangle$ digit to be placed after the $\langle X_i \rangle$, but the choice we make involves less reshuffling.

Note that this function fails for negative $\langle shift \rangle$.

```
11340 \cs_new:Npn \__fp_decimate:nNnnnn #1
11341   {
11342     \cs:w
11343       __fp_decimate_
11344       \if_int_compare:w \__int_eval:w #1 > \c_sixteen
11345         tiny
11346       \else:
11347         \__int_to_roman:w \__int_eval:w #1
11348       \fi:
11349       :Nnnnn
11350     \cs_end:
11351   }
```

Each of the auxiliaries see the function $\langle f_1 \rangle$, followed by 4 blocks of 4 digits.

(*End definition for* `\__fp_decimate:nNnnnn`.)

`\__fp_decimate_:Nnnnn`
`\__fp_decimate_tiny:Nnnnn`

If the $\langle shift \rangle$ is zero, or too big, life is very easy.

```
11352 \cs_new:Npn \__fp_decimate_:Nnnnn #1 #2#3#4#5
11353   { #1 0 {#2#3} {#4#5} ; }
11354 \cs_new:Npn \__fp_decimate_tiny:Nnnnn #1 #2#3#4#5
11355   { #1 1 { 0000 0000 } { 0000 0000 } 0 #2#3#4#5 ; }
```

(*End 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`

Shifting happens in two steps: compute the $\langle rounding \rangle$ 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 $\langle rounding \rangle$ digit. 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.

---

[10] No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

```
11356  \cs_new:Npn \__fp_tmp:w #1 #2 #3
11357    {
11358      \cs_new:cpn { __fp_decimate_ #1 :Nnnnn } ##1 ##2##3##4##5
11359        {
11360          \exp_after:wN ##1
11361          \__int_value:w
11362            \exp_after:wN \__fp_round_digit:Nw #2 ;
11363          \__fp_decimate_pack:nnnnnnnnnnw #3 ;
11364        }
11365    }
11366  \__fp_tmp:w {i}    {\use_none:nnn     #50}{    0{#2}#3{#4}#5                    }
11367  \__fp_tmp:w {ii}   {\use_none:nn      #5 }{    00{#2}#3{#4}#5                   }
11368  \__fp_tmp:w {iii}  {\use_none:n       #5 }{    000{#2}#3{#4}#5                  }
11369  \__fp_tmp:w {iv}   {                  #5 }{    {0000}#2{#3}#4 #5                }
11370  \__fp_tmp:w {v}    {\use_none:nnn     #4#5 }{   0{0000}#2{#3}#4 #5              }
11371  \__fp_tmp:w {vi}   {\use_none:nn      #4#5 }{   00{0000}#2{#3}#4 #5             }
11372  \__fp_tmp:w {vii}  {\use_none:n       #4#5 }{   000{0000}#2{#3}#4 #5            }
11373  \__fp_tmp:w {viii}{                   #4#5 }{   {0000}0000{#2}#3 #4 #5          }
11374  \__fp_tmp:w {ix}   {\use_none:nnn  #3#4+#5}{   0{0000}0000{#2}#3 #4 #5          }
11375  \__fp_tmp:w {x}    {\use_none:nn   #3#4+#5}{   00{0000}0000{#2}#3 #4 #5         }
11376  \__fp_tmp:w {xi}   {\use_none:n    #3#4+#5}{   000{0000}0000{#2}#3 #4 #5        }
11377  \__fp_tmp:w {xii} {               #3#4+#5}{   {0000}0000{0000}#2 #3 #4 #5      }
11378  \__fp_tmp:w {xiii}{\use_none:nnn#2#3+#4+#5}{ 0{0000}0000{0000}#2 #3 #4 #5      }
11379  \__fp_tmp:w {xiv} {\use_none:nn #2#3+#4+#5}{ 00{0000}0000{0000}#2 #3 #4 #5     }
11380  \__fp_tmp:w {xv}  {\use_none:n  #2#3+#4+#5}{ 000{0000}0000{0000}#2 #3 #4 #5    }
11381  \__fp_tmp:w {xvi} {               #2#3+#4+#5}{{0000}0000{0000}0000 #2 #3 #4 #5}
```

(*End definition for* `\__fp_decimate_auxi:Nnnnn` *and others.*)

`\__fp_round_digit:Nw`
`\__fp_decimate_pack:nnnnnnnnnnw`

`\__fp_round_digit:Nw` will receive the "extra digits" as its argument, and its expansion is triggered by `\__int_value:w`. If the first digit is neither 0 nor 5, then it is the ⟨*rounding*⟩ digit. Otherwise, if the remaining digits are not all zero, we need to add 1 to that leading digit to get the rounding digit. Some caution is required, though, because there may be more than 10 "extra digits", and this may overflow TEX's integers. Instead of feeding the digits directly to `\__fp_round_digit:Nw`, they come split into several blocks, separated by +. Hence the first `\__int_eval:w` here.

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.

```
11382  \cs_new:Npn \__fp_decimate_pack:nnnnnnnnnnw #1#2#3#4#5
11383    { \__fp_decimate_pack:nnnnnw { #1#2#3#4#5 } }
11384  \cs_new:Npn \__fp_decimate_pack:nnnnnw #1 #2#3#4#5#6
11385    { {#1} {#2#3#4#5#6} }
```

(*End definition for* `\__fp_round_digit:Nw` *and* `\__fp_decimate_pack:nnnnnnnnnnw`.)

22.9   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 In this example, the case 0 will return the floating point ⟨*fp var*⟩, expanding once after that floating point. Case 1 will do ⟨*some computation*⟩ using the ⟨*floating point*⟩ (presumably compute the operation requested by the user in that non-trivial case). Case 2 will return the ⟨*floating point*⟩ without modifying it, removing the ⟨*junk*⟩ and expanding once after. Case 3 will close the conditional, remove the ⟨*junk*⟩ and the ⟨*floating point*⟩, and expand ⟨*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.

```
11386 \cs_new:Npn \__fp_case_use:nw #1#2 \fi: #3 \s__fp { \fi: #1 \s__fp }
```

(*End 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.

```
11387 \cs_new:Npn \__fp_case_return:nw #1#2 \fi: #3 ; { \fi: #1 }
```

(*End 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.

```
11388 \cs_new:Npn \__fp_case_return_o:Nw #1#2 \fi: #3 \s__fp #4 ;
11389     { \fi: \exp_after:wN #1 }
```

(*End 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.

```
11390 \cs_new:Npn \__fp_case_return_same_o:w #1 \fi: #2 \s__fp
11391     { \fi: \__fp_exp_after_o:w \s__fp }
```

(*End 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.

```
11392 \cs_new:Npn \__fp_case_return_o:Nww #1#2 \fi: #3 \s__fp #4 ; #5 ;
11393     { \fi: \exp_after:wN #1 }
```

(*End 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.

```
11394 \cs_new:Npn \__fp_case_return_i_o:ww #1 \fi: #2 \s__fp #3 ; \s__fp #4 ;
11395     { \fi: \__fp_exp_after_o:w \s__fp #3 ; }
11396 \cs_new:Npn \__fp_case_return_ii_o:ww #1 \fi: #2 \s__fp #3 ;
11397     { \fi: \__fp_exp_after_o:w }
```

(*End definition for* __fp_case_return_i_o:ww *and* __fp_case_return_ii_o:ww.)

22.10 Integer floating points

Tests if the floating point argument is an integer. This holds if the rounding digit resulting from __fp_decimate:nNnnnn is 0.

```
11398 \prg_new_conditional:Npnn \__fp_int:w \s__fp \__fp_chk:w #1 #2 #3; { TF , T , F , p }
11399   {
11400     \if_case:w #1 \exp_stop_f:
11401             \prg_return_true:
11402     \or:   \__fp_int_normal:nnnnn #3
11403     \else: \prg_return_false:
11404     \fi:
11405   }
11406 \cs_new:Npn \__fp_int_normal:nnnnn #1 #2#3#4#5
11407   {
11408     \if_int_compare:w #1 > \c_zero
11409       \__fp_decimate:nNnnnn { \c_sixteen - #1 }
11410         \__fp_int_test:Nw
11411         {#2} {#3} {#4} {#5}
11412     \else:
11413       \prg_return_false:
11414     \fi:
11415   }
11416 \cs_new:Npn \__fp_int_test:Nw #1#2;
11417   {
11418     \if_meaning:w 0 #1
11419       \prg_return_true:
11420     \else:
11421       \prg_return_false:
11422     \fi:
11423   }
```

(*End definition for* __fp_int:wTF, __fp_int_normal:nnnnn, *and* __fp_int_test:Nw.)

22.11 Small integer floating points

Tests if the floating point argument is an integer or $\pm\infty$. If so, it is converted 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: neither nan nor infinities are integers. Normal numbers with a non-positive exponent are never integers. When the exponent is greater than 8, the number is too large for the range. Otherwise, decimate, and test the digits after the decimal separator. The \use_iii:nnn remove a trailing ; and the true branch, leaving only the false branch. The __int_value:w appearing in the case where the normal floating point is an integer takes care of expanding all the conditionals until the trailing ;. That integer is fed to __fp_small_int_true:wTF which places it as a braced argument of the true branch. The \use_i:nn in __fp_small_int_test:NnnwNTF removes the top-level \else: coming from __fp_small_int_normal:NnwTF, hence will call the \use_iii:nnn which follows, taking the false branch.

```
11424 \cs_new:Npn \__fp_small_int:wTF \s__fp \__fp_chk:w #1#2
11425   {
11426     \if_case:w #1 \exp_stop_f:
11427             \__fp_case_return:nw { \__fp_small_int_true:wTF 0 ; }
11428     \or:   \exp_after:wN \__fp_small_int_normal:NnwTF
11429     \or:
```

554

```
11430        \__fp_case_return:nw
11431          {
11432            \exp_after:wN \__fp_small_int_true:wTF \__int_value:w
11433              \if_meaning:w 2 #2 - \fi: 1 0000 0000 ;
11434          }
11435      \else: \__fp_case_return:nw \use_ii:nn
11436      \fi:
11437      #2
11438    }
11439 \cs_new:Npn \__fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
11440 \cs_new:Npn \__fp_small_int_normal:NnwTF #1#2#3;
11441    {
11442      \if_int_compare:w #2 > \c_zero
11443        \__fp_decimate:nNnnnn { \c_sixteen - #2 }
11444          \__fp_small_int_test:NnnwNnw
11445          #3 #1 {#2}
11446      \else:
11447        \exp_after:wN \use_iii:nnn
11448      \fi:
11449      ;
11450    }
11451 \cs_new:Npn \__fp_small_int_test:NnnwNnw #1#2#3#4; #5#6
11452    {
11453      \if_meaning:w 0 #1
11454        \exp_after:wN \__fp_small_int_true:wTF
11455        \__int_value:w \if_meaning:w 2 #5 - \fi:
11456          \if_int_compare:w #6 > \c_eight
11457            1 0000 0000
11458          \else:
11459            #3
11460          \fi:
11461      \else:
11462        \use_i:nn
11463      \fi:
11464    }
```

(*End definition for* `\__fp_small_int:wTF` *and others.*)

22.12   Length of a floating point array

`\__fp_array_count:n`
`\__fp_array_count_loop:Nw`

Count the number of items in an array of floating points. 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.

```
11465 \cs_new:Npn \__fp_array_count:n #1
11466    {
11467      \__int_value:w \__int_eval:w \c_zero
11468        \__fp_array_count_loop:Nw #1 { ? \__prg_break: } ;
11469        \__prg_break_point:
11470      \__int_eval_end:
11471    }
11472 \cs_new:Npn \__fp_array_count_loop:Nw #1#2;
11473    { \use_none:n #1 + \c_one \__fp_array_count_loop:Nw }
```

(*End definition for* `\__fp_array_count:n` *and* `\__fp_array_count_loop:Nw`.)

22.13 x-like expansion expandably

__fp_expand:n

__fp_expand_loop:nwnN

This expandable function behaves in a way somewhat similar to `\use:x`, but much less robust. The argument is f-expanded, then the leading item (often a single character token) is moved to a storage area after `\s__fp_mark`, and f-expansion is applied again, repeating until the argument is empty. The result built one piece at a time is then inserted in the input stream. Note that spaces are ignored by this procedure, unless surrounded with braces. Multiple tokens which do not need expansion can be inserted within braces.

```
11474 \cs_new:Npn \__fp_expand:n #1
11475   {
11476     \__fp_expand_loop:nwnN { }
11477       #1 \prg_do_nothing:
11478     \s__fp_mark { } \__fp_expand_loop:nwnN
11479     \s__fp_mark { } \__fp_use_i_until_s:nw ;
11480   }
11481 \cs_new:Npn \__fp_expand_loop:nwnN #1#2 \s__fp_mark #3 #4
11482   {
11483     \exp_after:wN #4 \exp:w \exp_end_continue_f:w
11484     #2
11485     \s__fp_mark { #3 #1 } #4
11486   }
```

(*End definition for* `\__fp_expand:n` *and* `\__fp_expand_loop:nwnN`.)

22.14 Messages

Using a floating point directly is an error.

```
11487 \__msg_kernel_new:nnnn { kernel } { misused-fp }
11488   { A~floating~point~with~value~'#1'~was~misused. }
11489   {
11490     To~obtain~the~value~of~a~floating~point~variable,~use~
11491     '\token_to_str:N \fp_to_decimal:N',~
11492     '\token_to_str:N \fp_to_scientific:N',~or~other~
11493     conversion~functions.
11494   }
11495 ⟨/initex | package⟩
```

23 l3fp-traps Implementation

```
11496 ⟨*initex | package⟩
11497 ⟨@@=fp⟩
```

Exceptions should be accessed by an n-type argument, among

- `invalid_operation`

- `division_by_zero`

- `overflow`

- `underflow`

- `inexact` (actually never used).

23.1 Flags

\fp_flag_off:n   Function to turn a flag off. Simply undefine it.

```
11498 \cs_new_protected:Npn \fp_flag_off:n #1
11499   { \cs_set_eq:cN { l__fp_ #1 _flag_token } \tex_undefined:D }
```

(*End definition for* `\fp_flag_off:n`*. This function is documented on page* *188.*)

\fp_flag_on:n   Function to turn a flag on expandably: use TEX's automatic assignment to \scan_stop:.

```
11500 \cs_new:Npn \fp_flag_on:n #1
11501   { \exp_args:Nc \use_none:n { l__fp_ #1 _flag_token } }
```

(*End definition for* `\fp_flag_on:n`*. This function is documented on page* *188.*)

\fp_if_flag_on_p:n   Returns true if the flag is on, false otherwise.
\fp_if_flag_on:n*TF*

```
11502 \prg_new_conditional:Npnn \fp_if_flag_on:n #1 { p , T , F , TF }
11503   {
11504     \if_cs_exist:w l__fp_ #1 _flag_token \cs_end:
11505       \prg_return_true:
11506     \else:
11507       \prg_return_false:
11508     \fi:
11509   }
```

(*End definition for* `\fp_if_flag_on:nTF`*. This function is documented on page* *188.*)

\l_fp_invalid_operation_flag_token   The IEEE standard defines five exceptions. We currently don't support the "inexact"
\l_fp_division_by_zero_flag_token   exception.
\l__fp_overflow_flag_token
\l__fp_underflow_flag_token

```
11510 \cs_new_eq:NN \l__fp_invalid_operation_flag_token \tex_undefined:D
11511 \cs_new_eq:NN \l__fp_division_by_zero_flag_token \tex_undefined:D
11512 \cs_new_eq:NN \l__fp_overflow_flag_token \tex_undefined:D
11513 \cs_new_eq:NN \l__fp_underflow_flag_token \tex_undefined:D
```

(*End definition for* `\l__fp_invalid_operation_flag_token` *and others.*)

23.2 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

```
11514 \cs_new_protected:Npn \fp_trap:nn #1#2
11515   {
11516     \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
11517       {
11518         \clist_if_in:nnTF
11519           { invalid_operation , division_by_zero , overflow , underflow }
11520           {#1}
11521           {
11522             \__msg_kernel_error:nnxx { kernel }
11523               { unknown-fpu-trap-type } {#1} {#2}
11524           }
11525           {
11526             \__msg_kernel_error:nnx
11527               { kernel } { unknown-fpu-exception } {#1}
11528           }
11529       }
11530   }
```

(*End definition for* \fp_trap:nn. *This function is documented on page 188.*)

__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.

```
11531 \cs_new_protected:Npn \__fp_trap_invalid_operation_set_error:
11532   { \__fp_trap_invalid_operation_set:N \prg_do_nothing: }
11533 \cs_new_protected:Npn \__fp_trap_invalid_operation_set_flag:
11534   { \__fp_trap_invalid_operation_set:N \use_none:nnnnn }
11535 \cs_new_protected:Npn \__fp_trap_invalid_operation_set_none:
11536   { \__fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
11537 \cs_new_protected:Npn \__fp_trap_invalid_operation_set:N #1
11538   {
11539     \exp_args:Nno \use:n
11540       { \cs_set:Npn \__fp_invalid_operation:nnw ##1##2##3; }
11541       {
11542         #1
11543         \__fp_error:nnfn { invalid } {##2} { \fp_to_tl:n { ##3; } } { }
11544         \fp_flag_on:n { invalid_operation }
11545         ##1
11546       }
11547     \exp_args:Nno \use:n
11548       { \cs_set:Npn \__fp_invalid_operation_o:Nww ##1##2; ##3; }
```

```
11549        {
11550          #1
11551          \__fp_error:nffn { invalid-ii }
11552            { \fp_to_tl:n { ##2; } } } { \fp_to_tl:n { ##3; } } } {##1}
11553          \fp_flag_on:n { invalid_operation }
11554          \exp_after:wN \c_nan_fp
11555        }
11556      \exp_args:Nno \use:n
11557        { \cs_set:Npn \__fp_invalid_operation_tl_o:ff ##1##2 }
11558        {
11559          #1
11560          \__fp_error:nffn { invalid } {##1} {##2} { }
11561          \fp_flag_on:n { invalid_operation }
11562          \exp_after:wN \c_nan_fp
11563        }
11564    }
```

(*End definition for* `\__fp_trap_invalid_operation_set_error:` *and others.*)

<div style="float-left">

`\__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`

</div>

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.

```
11565 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_error:
11566   { \__fp_trap_division_by_zero_set:N \prg_do_nothing: }
11567 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_flag:
11568   { \__fp_trap_division_by_zero_set:N \use_none:nnnnn }
11569 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_none:
11570   { \__fp_trap_division_by_zero_set:N \use_none:nnnnnnn }
11571 \cs_new_protected:Npn \__fp_trap_division_by_zero_set:N #1
11572   {
11573     \exp_args:Nno \use:n
11574       { \cs_set:Npn \__fp_division_by_zero_o:Nnw ##1##2##3; }
11575       {
11576         #1
11577         \__fp_error:nnfn { zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
11578         \fp_flag_on:n { division_by_zero }
11579         \exp_after:wN ##1
11580       }
11581     \exp_args:Nno \use:n
11582       { \cs_set:Npn \__fp_division_by_zero_o:NNww ##1##2##3; ##4; }
11583       {
11584         #1
11585         \__fp_error:nffn { zero-div-ii }
11586           { \fp_to_tl:n { ##3; } } } { \fp_to_tl:n { ##4; } } } {##2}
11587         \fp_flag_on:n { division_by_zero }
11588         \exp_after:wN ##1
11589       }
11590   }
```

(*End definition for* `\__fp_trap_division_by_zero_set_error:` *and others.*)

<div style="float-left">

`\__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`

</div>

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.

```
11591 \cs_new_protected:Npn \__fp_trap_overflow_set_error:
11592   { \__fp_trap_overflow_set:N \prg_do_nothing: }
11593 \cs_new_protected:Npn \__fp_trap_overflow_set_flag:
11594   { \__fp_trap_overflow_set:N \use_none:nnnnn }
11595 \cs_new_protected:Npn \__fp_trap_overflow_set_none:
11596   { \__fp_trap_overflow_set:N \use_none:nnnnnnn }
11597 \cs_new_protected:Npn \__fp_trap_overflow_set:N #1
11598   { \__fp_trap_overflow_set:NnNn #1 { overflow } \__fp_inf_fp:N { inf } }
11599 \cs_new_protected:Npn \__fp_trap_underflow_set_error:
11600   { \__fp_trap_underflow_set:N \prg_do_nothing: }
11601 \cs_new_protected:Npn \__fp_trap_underflow_set_flag:
11602   { \__fp_trap_underflow_set:N \use_none:nnnnn }
11603 \cs_new_protected:Npn \__fp_trap_underflow_set_none:
11604   { \__fp_trap_underflow_set:N \use_none:nnnnnnn }
11605 \cs_new_protected:Npn \__fp_trap_underflow_set:N #1
11606   { \__fp_trap_overflow_set:NnNn #1 { underflow } \__fp_zero_fp:N { 0 } }
11607 \cs_new_protected:Npn \__fp_trap_overflow_set:NnNn #1#2#3#4
11608   {
11609     \exp_args:Nno \use:n
11610       { \cs_set:cpn { __fp_ #2 :w } \s__fp \__fp_chk:w ##1##2##3; }
11611       {
11612         #1
11613         \__fp_error:nffn
11614           { flow \if_meaning:w 1 ##1 -to \fi: }
11615           { \fp_to_tl:n { \s__fp \__fp_chk:w ##1##2##3; } }
11616           { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }
11617           {#2}
11618         \fp_flag_on:n {#2}
11619         #3 ##2
11620       }
11621   }
```

(*End definition for* __fp_trap_overflow_set_error: *and others.*)

__fp_invalid_operation:nnw Initialize the two control sequences (to log properly their existence). Then set invalid
__fp_invalid_operation_o:Nww operations to trigger an error, and division by zero, overflow, and underflow to act silently
__fp_invalid_operation_tl_o:ff on their flag.
__fp_division_by_zero_o:Nnw
__fp_division_by_zero_o:NNww
__fp_overflow:w
__fp_underflow:w

```
11622 \cs_new:Npn \__fp_invalid_operation:nnw #1#2#3; { }
11623 \cs_new:Npn \__fp_invalid_operation_o:Nww #1#2; #3; { }
11624 \cs_new:Npn \__fp_invalid_operation_tl_o:ff #1 #2 { }
11625 \cs_new:Npn \__fp_division_by_zero_o:Nnw #1#2#3; { }
11626 \cs_new:Npn \__fp_division_by_zero_o:NNww #1#2#3; #4; { }
11627 \cs_new:Npn \__fp_overflow:w { }
11628 \cs_new:Npn \__fp_underflow:w { }
11629 \fp_trap:nn { invalid_operation } { error }
11630 \fp_trap:nn { division_by_zero } { flag }
```

```
11631 \fp_trap:nn { overflow } { flag }
11632 \fp_trap:nn { underflow } { flag }
```

(*End 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.

```
11633 \cs_new:Npn \__fp_invalid_operation_o:nw
11634   { \__fp_invalid_operation:nnw { \exp_after:wN \c_nan_fp } }
11635 \cs_generate_variant:Nn \__fp_invalid_operation_o:nw { f }
```

(*End definition for* __fp_invalid_operation_o:nw.)

23.3 Errors

__fp_error:nnnn
__fp_error:nnfn
__fp_error:nffn

```
11636 \cs_new:Npn \__fp_error:nnnn #1
11637   { \__msg_kernel_expandable_error:nnnnn { kernel } { fp - #1 } }
11638 \cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff }
```

(*End definition for* __fp_error:nnnn.)

23.4 Messages

Some messages.

```
11639 \__msg_kernel_new:nnnn { kernel } { unknown-fpu-exception }
11640   {
11641     The~FPU~exception~'#1'~is~not~known:~
11642     that~trap~will~never~be~triggered.
11643   }
11644   {
11645     The~only~exceptions~to~which~traps~can~be~attached~are \\
11646     \iow_indent:n
11647       {
11648         * ~ invalid_operation \\
11649         * ~ division_by_zero \\
11650         * ~ overflow \\
11651         * ~ underflow
11652       }
11653   }
11654 \__msg_kernel_new:nnnn { kernel } { unknown-fpu-trap-type }
11655   { The~FPU~trap~type~'#2'~is~not~known. }
11656   {
11657     The~trap~type~must~be~one~of \\
11658     \iow_indent:n
11659       {
11660         * ~ error \\
11661         * ~ flag \\
11662         * ~ none
11663       }
11664   }
11665 \__msg_kernel_new:nnn { kernel } { fp-flow }
11666   { An ~ #3 ~ occurred. }
11667 \__msg_kernel_new:nnn { kernel } { fp-flow-to }
```

```
11668     { #1 ~ #3 ed ~ to ~ #2 . }
11669 \__msg_kernel_new:nnn { kernel } { fp-zero-div }
11670     { Division~by~zero~in~ #1 (#2) }
11671 \__msg_kernel_new:nnn { kernel } { fp-zero-div-ii }
11672     { Division~by~zero~in~ (#1) #3 (#2) }
11673 \__msg_kernel_new:nnn { kernel } { fp-invalid }
11674     { Invalid~operation~ #1 (#2) }
11675 \__msg_kernel_new:nnn { kernel } { fp-invalid-ii }
11676     { Invalid~operation~ (#1) #3 (#2) }
```

11677 ⟨/initex | package⟩

24 l3fp-round implementation

11678 ⟨*initex | package⟩

11679 ⟨@@=fp⟩

24.1 Rounding tools

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 \c_zero or \c_one depending on whether the final result should be rounded up or down.

- __fp_round:NNN ⟨sign⟩ ⟨digit₁⟩ ⟨digit₂⟩ can expand to \c_zero or \c_one.

- __fp_round_s:NNNw ⟨sign⟩ ⟨digit₁⟩ ⟨digit₂⟩ ⟨more digits⟩; can expand to \c_zero ; or \c_one ;.

- __fp_round_neg:NNN ⟨sign⟩ ⟨digit₁⟩ ⟨digit₂⟩ can expand to \c_zero or \c_one.

See implementation comments for details on the syntax.

If rounding the number $\langle final\ sign\rangle\langle digit_1\rangle.\langle digit_2\rangle$ to an integer rounds it towards zero (truncates it), this function expands to `\c_zero`, and otherwise to `\c_one`. Typically used within the scope of an `\__int_eval:w`, to add 1 if needed, and thereby round correctly. The result depends on the rounding mode.

It is very important that $\langle final\ sign\rangle$ be the final sign of the result. Otherwise, the result will be incorrect in the case of rounding towards $-\infty$ or towards $+\infty$. Also recall that $\langle final\ sign\rangle$ is 0 for positive, and 2 for negative.

By default, the functions below return `\c_zero`, but this is superseded by `\__fp_-round_return_one:`, which instead returns `\c_one`, expanding everything and removing `\c_zero` 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 $\langle digit_2\rangle$ is larger than 5, then round up. If it is less than 5, round down. If it is exactly 5, then round such that $\langle digit_1\rangle$ plus the result is even. In other words, round up if $\langle digit_1\rangle$ 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 $+\infty$.

```
11680 \cs_new:Npn \__fp_round_return_one:
11681   { \exp_after:wN \c_one \exp:w }
11682 \cs_new:Npn \__fp_round_to_ninf:NNN #1 #2 #3
11683   {
11684     \if_meaning:w 2 #1
11685       \if_int_compare:w #3 > \c_zero
11686         \__fp_round_return_one:
11687       \fi:
11688     \fi:
11689     \c_zero
11690   }
11691 \cs_new:Npn \__fp_round_to_zero:NNN #1 #2 #3 { \c_zero }
11692 \cs_new:Npn \__fp_round_to_pinf:NNN #1 #2 #3
11693   {
11694     \if_meaning:w 0 #1
11695       \if_int_compare:w #3 > \c_zero
11696         \__fp_round_return_one:
11697       \fi:
11698     \fi:
11699     \c_zero
11700   }
11701 \cs_new:Npn \__fp_round_to_nearest:NNN #1 #2 #3
11702   {
11703     \if_int_compare:w #3 > \c_five
11704       \__fp_round_return_one:
11705     \else:
11706       \if_meaning:w 5 #3
11707         \if_int_odd:w #2 \exp_stop_f:
11708           \__fp_round_return_one:
11709         \fi:
11710       \fi:
11711     \fi:
11712     \c_zero
11713   }
11714 \cs_new:Npn \__fp_round_to_nearest_ninf:NNN #1 #2 #3
11715   {
```

563

```
11716        \if_int_compare:w #3 > \c_five
11717            \__fp_round_return_one:
11718        \else:
11719            \if_meaning:w 5 #3
11720                \if_meaning:w 2 #1
11721                    \__fp_round_return_one:
11722                \fi:
11723            \fi:
11724        \fi:
11725        \c_zero
11726    }
11727 \cs_new:Npn \__fp_round_to_nearest_zero:NNN #1 #2 #3
11728    {
11729        \if_int_compare:w #3 > \c_five
11730            \__fp_round_return_one:
11731        \fi:
11732        \c_zero
11733    }
11734 \cs_new:Npn \__fp_round_to_nearest_pinf:NNN #1 #2 #3
11735    {
11736        \if_int_compare:w #3 > \c_five
11737            \__fp_round_return_one:
11738        \else:
11739            \if_meaning:w 5 #3
11740                \if_meaning:w 0 #1
11741                    \__fp_round_return_one:
11742                \fi:
11743            \fi:
11744        \fi:
11745        \c_zero
11746    }
11747 \cs_new_eq:NN \__fp_round:NNN \__fp_round_to_nearest:NNN
```

(*End definition for* `\__fp_round:NNN` *and others.*)

`\__fp_round_s:NNNw` Similar to `\__fp_round:NNN`, but with an extra semicolon, this function expands to `\c_zero` ; if rounding ⟨*final sign*⟩⟨*digit*⟩.⟨*more digits*⟩ to an integer truncates, and to `\c_one` ; otherwise. The ⟨*more digits*⟩ part must be a digit, followed by something that does not overflow a `\int_use:N \__int_eval:w` construction. The only relevant information about this piece is whether it is zero or not.

```
11748 \cs_new:Npn \__fp_round_s:NNNw #1 #2 #3 #4;
11749    {
11750        \exp_after:wN \__fp_round:NNN
11751        \exp_after:wN #1
11752        \exp_after:wN #2
11753        \__int_value:w \__int_eval:w
11754            \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
11755                            \if_meaning:w 5 #3 1 \fi:
11756                    \exp_stop_f:
11757            \if_int_compare:w \__int_eval:w #4 > \c_zero
11758                1 +
11759            \fi:
11760            \fi:
11761            #3
```

564

```
11762          ;
11763      }
```

(*End definition for* `\__fp_round_s:NNNw`.)

`\__fp_round_digit:Nw`   This function should always be called within an `\__int_value:w` or `\__int_eval:w` expansion; it may add an extra `\__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.

```
11764 \cs_new:Npn \__fp_round_digit:Nw #1 #2;
11765   {
11766     \if_int_odd:w \if_meaning:w 0 #1 \c_one \else:
11767                     \if_meaning:w 5 #1 \c_one \else:
11768                     \c_zero \fi: \fi:
11769       \if_int_compare:w \__int_eval:w #2 > \c_zero
11770         \__int_eval:w \c_one +
11771       \fi:
11772     \fi:
11773     #1
11774   }
```

(*End 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`

This expands to `\c_zero` or `\c_one` 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 `\c_one`. Otherwise, *i.e.*, if the result is rounded back to the first operand, then this function returns `\c_zero`.

It turns out that this negative "round to nearest" is identical to the positive one. And this is the default mode.

```
11775 \cs_new_eq:NN \__fp_round_to_ninf_neg:NNN \__fp_round_to_pinf:NNN
11776 \cs_new:Npn \__fp_round_to_zero_neg:NNN #1 #2 #3
11777   {
11778     \if_int_compare:w #3 > \c_zero
11779       \__fp_round_return_one:
11780     \fi:
11781     \c_zero
11782   }
11783 \cs_new_eq:NN \__fp_round_to_pinf_neg:NNN \__fp_round_to_ninf:NNN
11784 \cs_new_eq:NN \__fp_round_to_nearest_neg:NNN \__fp_round_to_nearest:NNN
11785 \cs_new_eq:NN \__fp_round_to_nearest_ninf_neg:NNN \__fp_round_to_nearest_pinf:NNN
11786 \cs_new:Npn \__fp_round_to_nearest_zero_neg:NNN #1 #2 #3
11787   {
11788     \if_int_compare:w #3 > \c_four
11789       \__fp_round_return_one:
11790     \fi:
11791     \c_zero
11792   }
11793 \cs_new_eq:NN \__fp_round_to_nearest_pinf_neg:NNN \__fp_round_to_nearest_ninf:NNN
11794 \cs_new_eq:NN \__fp_round_neg:NNN \__fp_round_to_nearest_neg:NNN
```

(*End definition for* `\__fp_round_neg:NNN` *and others.*)

24.2 The `round` function

`\__fp_round_o:Nw`   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 will change #1 from `\__fp_round_to_nearest:NNN` to one of its analogues.

```
11795 \cs_new:Npn \__fp_round_o:Nw #1#2 @
11796   {
11797     \if_case:w
11798       \__int_eval:w \__fp_array_count:n {#2} \__int_eval_end:
11799         \__fp_round_no_arg_o:Nw #1 \exp:w
11800     \or: \__fp_round:Nwn #1 #2 {0} \exp:w
11801     \or: \__fp_round:Nww #1 #2 \exp:w
11802     \else: \__fp_round:Nwww #1 #2 @ \exp:w
11803     \fi:
11804     \exp_end_continue_f:w
11805   }
```

(*End definition for* `\__fp_round_o:Nw`.)

`\__fp_round_no_arg_o:Nw`

```
11806 \cs_new:Npn \__fp_round_no_arg_o:Nw #1
11807   {
11808     \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
11809       { \__fp_error:nnnn { num-args } { round () } { 1 } { 3 } }
11810       {
11811         \__fp_error:nffn { num-args }
11812           { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
11813       }
11814     \exp_after:wN \c_nan_fp
11815   }
```

(*End 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.

```
11816 \cs_new:Npn \__fp_round:Nwww #1#2 ; #3 ; \s__fp \__fp_chk:w #4#5#6 ; #7 @
11817   {
11818     \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
11819       {
11820         \tl_if_empty:nTF {#7}
11821           {
11822             \exp_args:Nc \__fp_round:Nww
11823               {
11824                 __fp_round_to_nearest
11825                 \if_meaning:w 0 #4 _zero \else:
11826                 \if_case:w #5 \exp_stop_f: _pinf \or: \else: _ninf \fi: \fi:
11827                 :NNN
11828               }
11829             #2 ; #3 ;
11830           }
11831           {
11832             \__fp_error:nnnn { num-args } { round () } { 1 } { 3 }
```

```
11833                  \exp_after:wN \c_nan_fp
11834                }
11835            }
11836            {
11837              \__fp_error:nffn { num-args }
11838                { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
11839              \exp_after:wN \c_nan_fp
11840            }
11841        }
```

(*End definition for* __fp_round:Nwww.)

__fp_round_name_from_cs:N

```
11842 \cs_new:Npn \__fp_round_name_from_cs:N #1
11843    {
11844      \cs_if_eq:NNTF #1 \__fp_round_to_zero:NNN { trunc }
11845        {
11846          \cs_if_eq:NNTF #1 \__fp_round_to_ninf:NNN { floor }
11847            {
11848              \cs_if_eq:NNTF #1 \__fp_round_to_pinf:NNN { ceil }
11849                { round }
11850            }
11851        }
11852    }
```

(*End 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

```
11853 \cs_new:Npn \__fp_round:Nww #1#2 ; #3 ;
11854    {
11855      \__fp_small_int:wTF #3; { \__fp_round:Nwn #1#2; }
11856        {
11857          \__fp_invalid_operation_tl_o:ff
11858            { \__fp_round_name_from_cs:N #1 }
11859            { \__fp_array_to_clist:n { #2; #3; } }
11860        }
11861    }
11862 \cs_new:Npn \__fp_round:Nwn #1 \s__fp \__fp_chk:w #2#3#4; #5
11863    {
11864      \if_meaning:w 1 #2
11865        \exp_after:wN \__fp_round_normal:NwNNnw
11866        \exp_after:wN #1
11867        \__int_value:w #5
11868      \else:
11869        \exp_after:wN \__fp_exp_after_o:w
11870      \fi:
11871      \s__fp \__fp_chk:w #2#3#4;
11872    }
11873 \cs_new:Npn \__fp_round_normal:NwNNnw #1#2 \s__fp \__fp_chk:w 1#3#4#5;
11874    {
11875      \__fp_decimate:nNnnnn { \c_sixteen - #4 - #2 }
11876        \__fp_round_normal:NnnwNNnn #5 #1 #3 {#4} {#2}
11877    }
11878 \cs_new:Npn \__fp_round_normal:NnnwNNnn #1#2#3#4; #5#6
11879    {
```

```
11880    \exp_after:wN \__fp_round_normal:NNwNnn
11881    \__int_value:w \__int_eval:w
11882      \if_int_compare:w #2 > \c_zero
11883        1 \__int_value:w #2
11884        \exp_after:wN \__fp_round_pack:Nw
11885        \__int_value:w \__int_eval:w 1#3 +
11886      \else:
11887        \if_int_compare:w #3 > \c_zero
11888          1 \__int_value:w #3 +
11889        \fi:
11890      \fi:
11891      \exp_after:wN #5
11892      \exp_after:wN #6
11893      \use_none:nnnnnnn #3
11894      #1
11895      \__int_eval_end:
11896      0000 0000 0000 0000 ; #6
11897  }
11898 \cs_new:Npn \__fp_round_pack:Nw #1
11899    { \if_meaning:w 2 #1 + \c_one \fi: \__int_eval_end: }
11900 \cs_new:Npn \__fp_round_normal:NNwNnn #1 #2
11901    {
11902      \if_meaning:w 0 #2
11903        \exp_after:wN \__fp_round_special:NwwNnn
11904        \exp_after:wN #1
11905      \fi:
11906      \__fp_pack_twice_four:wNNNNNNNN
11907      \__fp_pack_twice_four:wNNNNNNNN
11908      \__fp_round_normal_end:wwNnn
11909      ; #2
11910    }
11911 \cs_new:Npn \__fp_round_normal_end:wwNnn #1;#2;#3#4#5
11912    {
11913      \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
11914      \__fp_sanitize:Nw #3 #4 ; #1 ;
11915    }
11916 \cs_new:Npn \__fp_round_special:NwwNnn #1#2;#3;#4#5#6
11917    {
11918      \if_meaning:w 0 #1
11919        \__fp_case_return:nw
11920          { \exp_after:wN \__fp_zero_fp:N \exp_after:wN #4 }
11921      \else:
11922        \exp_after:wN \__fp_round_special_aux:Nw
11923        \exp_after:wN #4
11924        \__int_value:w \__int_eval:w \c_one
11925          \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
11926      \fi:
11927      ;
11928    }
11929 \cs_new:Npn \__fp_round_special_aux:Nw #1#2;
11930    {
11931      \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
11932      \__fp_sanitize:Nw #1#2; {1000}{0000}{0000}{0000};
11933    }
```

(*End definition for* `\__fp_round:Nww` *and others.*)

25 l3fp-parse implementation

25.1 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.

`\__fp_parse:n` Evaluates the ⟨*floating point expression*⟩ and leaves the result in the input stream as an internal floating point number. 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 will lead to unrecoverable low-level TEX errors.

(*End definition for* `\__fp_parse:n`.)

   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 with multiple arguments.

15 Function calls expecting exactly one argument.

14 Binary ** and ^ (right to left).

12 Unary +, -, ! (right to left).

10 Binary *, /, and juxtaposition (implicit *).

 9 Binary + and -.

 7 Comparisons.

 5 Logical and, denoted by &&.

 4 Logical or, denoted by ||.

 3 Ternary operator ?:, piece ?.

 2 Ternary operator ?:, piece :.

 1 Commas, and parentheses accepting commas.

 0 Parentheses expecting exactly one argument.

-1 Start and end of the expression.

25.1.1 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 will grow 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 much 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 `\c_zero`, 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 \c_zero 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 `\c_zero`, 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 will 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.

25.1.2 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 the calculation `41-2^3*4+5` will be done. Here, we abuse notations: the first argument of `\operand:Nw` should be an integer constant (`\c_three`, `\c_nine`, ...) 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 will be 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*⟩ ⟨*number*₂⟩
> `@ \__fp_parse_infix_`⟨*operator*₂⟩`: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*⟩ ⟨*number*₂⟩
> @ `\__fp_parse_infix_`⟨*operator*₂⟩`: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*⟩ ⟨*number*₂⟩
> @ `\__fp_parse_infix_`⟨*operator*₂⟩`: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*⟩ ⟨*number*₂⟩
> `\exp:w \exp_end_continue_f:w`
> `\__fp_parse_infix_`⟨*operator*₂⟩`: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 will describe various subtleties.

25.1.3 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 comparision 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.

25.1.4 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.

574

- 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_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_-mark`, and otherwise it is a case of juxtaposing numbers, such as `2\c_three`, 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_three 2` is disallowed.

In the above, we need to test whether a character token `#1` is a digit:

```
\if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
  is a digit
\else:
  not a digit
\fi:
```

To exclude 0, replace `\c_nine` by `\c_ten`. 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 \__int_eval:w
    ( '#1 \if_int_compare:w '#1 > 'Z - 32 \fi: ) / 26 = \c_three
  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 $\{1, 2, 4, 10, 13\}$ will not work, 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 will not be expanded if we simply perform f-expansion.

```
\DeclareDocumentCommand {\test} {m} { \fp_eval:n {#1} }
\ExplSyntaxOff
\test { 1 + \X }
```

Of course, spaces will 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 will stop the f-expansion before further macro calls are performed. However, in practice it should be enough: in particular, floating point numbers will correctly be expanded to the underlying \s__fp ... structure. The f-expansion is performed by __fp_parse_expand:w.

25.2 Main auxiliary functions

__fp_parse_operand:Nw   Reads the "...", performing every computation with a precedence higher than ⟨*precedence*⟩, then expands to where the ⟨*operation*⟩ is the first operation with a lower precedence, possibly end, and the "..." start just after the ⟨*operation*⟩.

(*End definition for* __fp_parse_operand:Nw.)

__fp_parse_infix_+:N   If + has a precedence higher than the ⟨*precedence*⟩, cleans up a second ⟨*operand*⟩ and finds the ⟨*operation₂*⟩ which follows, and expands to Otherwise expands to A similar function exists for each infix operator.

(*End definition for* __fp_parse_infix_+:N.)

__fp_parse_one:Nw   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 and otherwise expands to

(*End definition for* __fp_parse_one:Nw.)

25.3 Helpers

__fp_parse_expand:w

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.

```
11937 \cs_new:Npn \__fp_parse_expand:w #1 { \exp_end_continue_f:w #1 }
```

(*End 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.

```
11938 \cs_new:Npn \__fp_parse_return_semicolon:w
11939     #1 \fi: \__fp_parse_expand:w { \fi: ; #1 }
```

(*End definition for* __fp_parse_return_semicolon:w.)

__fp_type_from_scan:N
__fp_type_from_scan:w

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 _?.

```
11940 \cs_new:Npx \__fp_type_from_scan:N #1
11941   {
11942     \exp_not:N \exp_after:wN \exp_not:N \__fp_type_from_scan:w
11943     \exp_not:N \token_to_str:N #1 \exp_not:N \q_mark
11944       \tl_to_str:n { s__fp _? } \exp_not:N \q_mark \exp_not:N \q_stop
11945   }
11946 \use:x
11947   {
11948     \cs_new:Npn \exp_not:N \__fp_type_from_scan:w
11949       ##1 \tl_to_str:n { s__fp } ##2 \exp_not:N \q_mark ##3 \exp_not:N \q_stop
11950       {##2}
11951   }
```

(*End definition for* __fp_type_from_scan:N *and* __fp_type_from_scan: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 __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.

```
11952 \cs_set_protected:Npn \__fp_tmp:w #1 #2 #3
11953   {
11954     \cs_new:cpn { __fp_parse_digits_ #1 :N } ##1
11955       {
11956         \if_int_compare:w \c_nine < 1 \token_to_str:N ##1 \exp_stop_f:
11957           \token_to_str:N ##1 \exp_after:wN #2 \exp_w
11958         \else:
11959           \__fp_parse_return_semicolon:w #3 ##1
```

577

```
11960          \fi:
11961          \__fp_parse_expand:w
11962        }
11963    }
11964 \__fp_tmp:w {vii}  \__fp_parse_digits_vi:N   { 0000000 ; 7 }
11965 \__fp_tmp:w {vi}   \__fp_parse_digits_v:N    { 000000 ; 6 }
11966 \__fp_tmp:w {v}    \__fp_parse_digits_iv:N   { 00000 ; 5 }
11967 \__fp_tmp:w {iv}   \__fp_parse_digits_iii:N  { 0000 ; 4 }
11968 \__fp_tmp:w {iii}  \__fp_parse_digits_ii:N   { 000 ; 3 }
11969 \__fp_tmp:w {ii}   \__fp_parse_digits_i:N    { 00 ; 2 }
11970 \__fp_tmp:w {i}    \__fp_parse_digits_:N     { 0 ; 1 }
11971 \cs_new:Npn \__fp_parse_digits_:N { ; ; 0 }
```

(*End definition for* `\__fp_parse_digits_vii:N` *and others.*)

25.4   Parsing one number

`\__fp_parse_one:Nw`

This function finds one number, and packs the symbol which follows in an `\..._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 will be split further). Despite the earlier f-expansion, `#2` may still be expandable if it was protected by `\exp_not:N`, as may happen with the LaTeX 2$_\varepsilon$ command `\protect`. Using a well placed `\reverse_if:N`, this case is sent to `\__fp_parse_one_fp:NN` which deals with it robustly.

```
11972 \cs_new:Npn \__fp_parse_one:Nw #1 #2
11973    {
11974      \if_catcode:w \scan_stop: \exp_not:N #2
11975        \exp_after:wN \if_meaning:w \exp_not:N #2 #2 \else:
11976          \exp_after:wN \reverse_if:N
11977        \fi:
11978        \if_meaning:w \scan_stop: #2
11979          \exp_after:wN \exp_after:wN
11980          \exp_after:wN \__fp_parse_one_fp:NN
11981        \else:
11982          \exp_after:wN \exp_after:wN
11983          \exp_after:wN \__fp_parse_one_register:NN
11984        \fi:
11985      \else:
11986        \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
11987          \exp_after:wN \exp_after:wN
11988          \exp_after:wN \__fp_parse_one_digit:NN
11989        \else:
11990          \exp_after:wN \exp_after:wN
11991          \exp_after:wN \__fp_parse_one_other:NN
11992        \fi:
11993      \fi:
11994      #1 #2
11995    }
```

(*End definition for* `\__fp_parse_one:Nw.`)

`\__fp_parse_one_fp:NN`
`\__fp_exp_after_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, dispatched using `\__fp_type_from_scan:N`.

- \s__fp starts a floating point number, and we call __fp_exp_after_f:nw, which f-expands after the floating point.

- \s__fp_mark is a premature end, we call __fp_exp_after_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 2ε uses \@unexpandable@protect rather than \protect because \protect is often \scan_stop: hence "does not exist".

```
11996 \cs_new:Npn \__fp_parse_one_fp:NN #1#2
11997   {
11998     \cs:w __fp_exp_after \__fp_type_from_scan:N #2 _f:nw \cs_end:
11999       {
12000         \exp_after:wN \__fp_parse_infix:NN
12001         \exp_after:wN #1 \exp:w \__fp_parse_expand:w
12002       }
12003     #2
12004   }
12005 \cs_new:Npn \__fp_exp_after_mark_f:nw #1
12006   {
12007     \__msg_kernel_expandable_error:nn { kernel } { fp-early-end }
12008     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
12009   }
12010 \cs_new:cpn { __fp_exp_after_?_f:nw } #1#2
12011   {
12012     \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2}
12013     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
12014   }
12015 ⟨*package⟩
12016 \group_begin:
12017   \char_set_catcode_letter:N \@
12018   \cs_if_exist:NT \@unexpandable@protect
12019     {
12020       \cs_gset:cpn { __fp_exp_after_?_f:nw } #1#2
12021         {
12022           \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
12023           \str_if_eq:nnTF {#2} { \protect }
12024             {
12025               \cs_if_eq:NNTF #2 \@unexpandable@protect { \use_i:nn } { \use:n }
12026               { \__msg_kernel_expandable_error:nnn { kernel } { fp-robust-cmd } }
12027             }
12028             { \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2} }
12029         }
12030     }
12031 \group_end:
12032 ⟨/package⟩
```

__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 assume that it is a register, but carefully unpacking 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_eval:w ⟨*decimal value*⟩ pt, and use an auxiliary of \dim_to_fp:n, which performs the multiplication by 2^{-16}, correctly rounded.

```
12033 \cs_new:Npn \__fp_parse_one_register:NN #1#2
12034   {
12035     \exp_after:wN \__fp_parse_infix_after_operand:NwN
12036     \exp_after:wN #1
12037     \exp:w \exp_end_continue_f:w
12038       \exp_after:wN \__fp_parse_one_register_aux:Nw
12039       \exp_after:wN #2
12040       \__int_value:w
12041         \exp_after:wN \__fp_parse_exponent:N
12042         \exp:w \__fp_parse_expand:w
12043   }
12044 \cs_new:Npx \__fp_parse_one_register_aux:Nw #1
12045   {
12046     \exp_not:n
12047       {
12048         \exp_after:wN \use:nn
12049         \exp_after:wN \__fp_parse_one_register_auxii:wwwNw
12050       }
12051     \exp_not:N \exp_after:wN { \exp_not:N \tex_the:D #1 }
12052     ; \exp_not:N \__fp_parse_one_register_dim:ww
12053     \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_mu:www
12054     . \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_int:www
12055     \exp_not:N \q_stop
12056   }
12057 \use:x
12058   {
12059     \cs_new:Npn \exp_not:N \__fp_parse_one_register_auxii:wwwNw
12060       ##1 . ##2 \tl_to_str:n { pt } ##3 ; ##4##5 \exp_not:N \q_stop
12061       { ##4 ##1.##2; }
12062     \cs_new:Npn \exp_not:N \__fp_parse_one_register_mu:www
12063       ##1 \tl_to_str:n { mu } ; ##2 ;
12064       { \exp_not:N \__fp_parse_one_register_dim:ww ##1 ; }
12065   }
12066 \cs_new:Npn \__fp_parse_one_register_int:www #1; #2.; #3;
12067   { \__fp_parse:n { #1 e #3 } }
12068 \cs_new:Npn \__fp_parse_one_register_dim:ww #1; #2;
12069   {
12070     \exp_after:wN \__fp_from_dim_test:ww
12071     \__int_value:w #2 \exp_after:wN ,
12072     \__int_value:w \__dim_eval:w #1 pt ;
```

$\_\_fp\_parse\_one\_digit:NN$ A digit marks the beginning of an explicit floating point number. Once the number is found, we will 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.

```
12074 \cs_new:Npn \__fp_parse_one_digit:NN #1
12075   {
12076     \exp_after:wN \__fp_parse_infix_after_operand:NwN
12077     \exp_after:wN #1
12078     \exp:w \exp_end_continue_f:w
12079       \exp_after:wN \__fp_sanitize:wN
12080       \__int_value:w \__int_eval:w \c_zero \__fp_parse_trim_zeros:N
12081   }
```

(*End 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 a 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 $\_\_fp\_parse\_prefix\_\langle operator\rangle:Nw$.

```
12082 \cs_new:Npn \__fp_parse_one_other:NN #1 #2
12083   {
12084     \if_int_compare:w
12085         \__int_eval:w
12086           ( `#2 \if_int_compare:w `#2 > `Z - \c_thirty_two \fi: ) / 26
12087         = \c_three
12088       \exp_after:wN \__fp_parse_word:Nw
12089       \exp_after:wN #1
12090       \exp_after:wN #2
12091       \exp:w \exp_after:wN \__fp_parse_letters:N
12092       \exp:w
12093     \else:
12094       \exp_after:wN \__fp_parse_prefix:NNN
12095       \exp_after:wN #1
12096       \exp_after:wN #2
12097       \cs:w
12098         __fp_parse_prefix_ \token_to_str:N #2 :Nw
12099         \exp_after:wN
12100       \cs_end:
12101       \exp:w
12102     \fi:
12103     \__fp_parse_expand:w
12104   }
```

(*End 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.

```
12105 \cs_new:Npn \__fp_parse_word:Nw #1#2;
12106   {
12107     \cs_if_exist_use:cF { __fp_parse_word_#2:N }
12108       {
12109         \__msg_kernel_expandable_error:nnn
12110           { kernel } { unknown-fp-word } {#2}
12111         \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
12112         \__fp_parse_infix:NN
12113       }
12114     #1
12115   }
12116 \cs_new:Npn \__fp_parse_letters:N #1
12117   {
12118     \exp_end_continue_f:w
12119     \if_int_compare:w
12120       \if_catcode:w \scan_stop: \exp_not:N #1
12121         \c_zero
12122       \else:
12123         \__int_eval:w
12124           ( '#1 \if_int_compare:w '#1 > 'Z - \c_thirty_two \fi: )
12125           / 26
12126       \fi:
12127       = \c_three
12128       \exp_after:wN #1
12129       \exp:w \exp_after:wN \__fp_parse_letters:N
12130       \exp:w
12131     \else:
12132       \__fp_parse_return_semicolon:w #1
12133     \fi:
12134     \__fp_parse_expand:w
12135   }
```

(*End 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.

```
12136 \cs_new:Npn \__fp_parse_prefix:NNN #1#2#3
12137   {
12138     \if_meaning:w \scan_stop: #3
12139       \exp_after:wN \__fp_parse_prefix_unknown:NNN
12140       \exp_after:wN #2
12141     \fi:
12142     #3 #1
12143   }
12144 \cs_new:Npn \__fp_parse_prefix_unknown:NNN #1#2#3
12145   {
```

```
12146        \cs_if_exist:cTF { __fp_parse_infix_ \token_to_str:N #1 :N }
12147          {
12148            \__msg_kernel_expandable_error:nnn
12149              { kernel } { fp-missing-number } {#1}
12150            \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
12151            \__fp_parse_infix:NN #3 #1
12152          }
12153          {
12154            \__msg_kernel_expandable_error:nnn
12155              { kernel } { fp-unknown-symbol } {#1}
12156            \__fp_parse_one:Nw #3
12157          }
12158      }
```

(*End definition for* `\__fp_parse_prefix:NNN` *and* `\__fp_parse_prefix_unknown:NNN`.)

25.4.1 Numbers: trimming leading zeros

Numbers will be 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.

```
12159 \cs_new:Npn \__fp_parse_trim_zeros:N #1
12160   {
12161     \if:w 0 \exp_not:N #1
12162       \exp_after:wN \__fp_parse_trim_zeros:N
12163       \exp:w
12164     \else:
12165       \if:w . \exp_not:N #1
12166         \exp_after:wN \__fp_parse_strim_zeros:N
12167         \exp:w
12168       \else:
12169         \__fp_parse_trim_end:w #1
12170       \fi:
12171     \fi:
12172     \__fp_parse_expand:w
12173   }
12174 \cs_new:Npn \__fp_parse_trim_end:w #1 \fi: \fi: \__fp_parse_expand:w
12175   {
12176       \fi:
12177       \fi:
12178     \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
12179       \exp_after:wN \__fp_parse_large:N
12180     \else:
12181       \exp_after:wN \__fp_parse_zero:
```

583

```
12182        \fi:
12183        #1
12184      }
```

(*End 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".

```
12185  \cs_new:Npn \__fp_parse_strim_zeros:N #1
12186    {
12187      \if:w 0 \exp_not:N #1
12188        - \c_one
12189        \exp_after:wN \__fp_parse_strim_zeros:N \exp:w
12190      \else:
12191        \__fp_parse_strim_end:w #1
12192      \fi:
12193      \__fp_parse_expand:w
12194    }
12195  \cs_new:Npn \__fp_parse_strim_end:w #1 \fi: \__fp_parse_expand:w
12196    {
12197      \fi:
12198      \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
12199        \exp_after:wN \__fp_parse_small:N
12200      \else:
12201        \exp_after:wN \__fp_parse_zero:
12202      \fi:
12203      #1
12204    }
```

(*End definition for* `\__fp_parse_strim_zeros:N` *and* `\__fp_parse_strim_end:w`.)

`\__fp_parse_zero:`

After reading a significand of 0, we need to remove any exponent, then put a sign of 1 for `\__fp_sanitize:wN`, small hack to denote an exact zero (rather than an underflow).

```
12205  \cs_new:Npn \__fp_parse_zero:
12206    {
12207      \exp_after:wN ; \exp_after:wN 1
12208      \__int_value:w \__fp_parse_exponent:N
12209    }
```

(*End definition for* `\__fp_parse_zero:`.)

25.4.2 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 will leave those digits in the `\__int_value:w`, and grab some more, or stop if there are no more digits.

584

Then the `pack_leading` auxiliary puts the various parts in the appropriate order for the processing further up.

```
12210 \cs_new:Npn \__fp_parse_small:N #1
12211   {
12212     \exp_after:wN \__fp_parse_pack_leading:NNNNNww
12213     \__int_value:w \__int_eval:w 1 \token_to_str:N #1
12214       \exp_after:wN \__fp_parse_small_leading:wwNN
12215       \__int_value:w 1
12216         \exp_after:wN \__fp_parse_digits_vii:N
12217         \exp:w \__fp_parse_expand:w
12218   }
```

(*End definition for* `\__fp_parse_small:N`.)

`\__fp_parse_small_leading:wwNN`   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 `\c_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.

```
12219 \cs_new:Npn \__fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
12220   {
12221     #1 #2
12222     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
12223     \exp_after:wN \c_zero
12224     \__int_value:w \__int_eval:w 1
12225       \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
12226         \token_to_str:N #4
12227         \exp_after:wN \__fp_parse_small_trailing:wwNN
12228         \__int_value:w 1
12229           \exp_after:wN \__fp_parse_digits_vi:N
12230           \exp:w
12231       \else:
12232         0000 0000 \__fp_parse_exponent:Nw #4
12233       \fi:
12234       \__fp_parse_expand:w
12235   }
```

(*End definition for* `\__fp_parse_small_leading:wwNN`.)

`\__fp_parse_small_trailing:wwNN`   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 `+\c_zero` or to `+\c_one`. Otherwise, there is no 16-th digit, so we put a 0, and look for an exponent.

```
12236 \cs_new:Npn \__fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
12237   {
12238     #1 #2
12239     \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
12240       \token_to_str:N #4
12241       \exp_after:wN \__fp_parse_small_round:NN
12242       \exp_after:wN #4
12243       \exp:w
```

```
12244        \else:
12245          0 \__fp_parse_exponent:Nw #4
12246        \fi:
12247        \__fp_parse_expand:w
12248      }
```

(*End 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 (`+ \c_one` 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.

```
12249 \cs_new:Npn \__fp_parse_pack_trailing:NNNNNNww #1 #2 #3#4#5#6 #7; #8 ;
12250    {
12251      \if_meaning:w 2 #2 + \c_one \fi:
12252      ; #8 + #1 ; {#3#4#5#6} {#7};
12253    }
12254 \cs_new:Npn \__fp_parse_pack_leading:NNNNNww #1 #2#3#4#5 #6; #7;
12255    {
12256      + #7
12257      \if_meaning:w 2 #1 \__fp_parse_pack_carry:w \fi:
12258      ; 0 {#2#3#4#5} {#6}
12259    }
12260 \cs_new:Npn \__fp_parse_pack_carry:w \fi: ; 0 #1
12261    { \fi: + \c_one ; 0 {1000} }
```

(*End definition for* `\__fp_parse_pack_trailing:NNNNNNww`, `\__fp_parse_pack_leading:NNNNNww`, *and* `\__fp_parse_pack_carry:w`.)

25.4.3 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.

```
12262 \cs_new:Npn \__fp_parse_large:N #1
12263    {
12264      \exp_after:wN \__fp_parse_large_leading:wwNN
12265      \__int_value:w 1 \token_to_str:N #1
12266        \exp_after:wN \__fp_parse_digits_vii:N
12267        \exp:w \__fp_parse_expand:w
12268    }
```

(*End definition for* `\__fp_parse_large:N`.)

_fp_parse_large_leading:wwNN 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.

```
12269 \cs_new:Npn \__fp_parse_large_leading:wwNN 1 #1 ; #2; #3 #4
12270   {
12271     + \c_eight - #3
12272     \exp_after:wN \__fp_parse_pack_leading:NNNNNww
12273     \__int_value:w \__int_eval:w 1 #1
12274       \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
12275         \exp_after:wN \__fp_parse_large_trailing:wwNN
12276         \__int_value:w 1 \token_to_str:N #4
12277           \exp_after:wN \__fp_parse_digits_vi:N
12278           \exp:w
12279       \else:
12280         \if:w . \exp_not:N #4
12281           \exp_after:wN \__fp_parse_small_leading:wwNN
12282           \__int_value:w 1
12283             \cs:w
12284               __fp_parse_digits_
12285               \__int_to_roman:w #3
12286               :N \exp_after:wN
12287             \cs_end:
12288             \exp:w
12289         \else:
12290           #2
12291           \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
12292           \exp_after:wN \c_zero
12293           \__int_value:w 1 0000 0000
12294           \__fp_parse_exponent:Nw #4
12295         \fi:
12296       \fi:
12297       \__fp_parse_expand:w
12298   }
```

(*End definition for* __fp_parse_large_leading:wwNN.)

_fp_parse_large_trailing:wwNN 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.

```
12299 \cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
12300   {
12301     \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
12302       \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
12303       \exp_after:wN \c_eight
```

```
12304         \__int_value:w \__int_eval:w 1 #1 \token_to_str:N #4
12305           \exp_after:wN \__fp_parse_large_round:NN
12306           \exp_after:wN #4
12307           \exp:w
12308       \else:
12309         \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
12310         \__int_value:w \__int_eval:w \c_seven - #3 \exp_stop_f:
12311         \__int_value:w \__int_eval:w 1 #1
12312           \if:w . \exp_not:N #4
12313             \exp_after:wN \__fp_parse_small_trailing:wwNN
12314             \__int_value:w 1
12315               \cs:w
12316                 __fp_parse_digits_
12317                 \__int_to_roman:w #3
12318                 :N \exp_after:wN
12319               \cs_end:
12320               \exp:w
12321           \else:
12322             #2 0 \__fp_parse_exponent:Nw #4
12323           \fi:
12324       \fi:
12325       \__fp_parse_expand:w
12326     }
```

(End definition for `\__fp_parse_large_trailing:wwNN`.*)*

25.4.4 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 `;\c_zero`, otherwise by `;\c_one`. 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.

```
12327 \cs_new:Npn \__fp_parse_round_loop:N #1
12328   {
12329     \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
12330       + \c_one
12331       \if:w 0 \token_to_str:N #1
12332         \exp_after:wN \__fp_parse_round_loop:N
12333         \exp:w
12334       \else:
12335         \exp_after:wN \__fp_parse_round_up:N
12336         \exp:w
12337       \fi:
12338     \else:
12339       \__fp_parse_return_semicolon:w \c_zero #1
12340     \fi:
12341     \__fp_parse_expand:w
12342   }
12343 \cs_new:Npn \__fp_parse_round_up:N #1
12344   {
12345     \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
12346       + \c_one
```

```
12347       \exp_after:wN \__fp_parse_round_up:N
12348       \exp:w
12349     \else:
12350       \__fp_parse_return_semicolon:w \c_one #1
12351     \fi:
12352     \__fp_parse_expand:w
12353   }
```

(*End 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 `\c_zero` or `\c_one` is added to the surrounding integer expression.

```
12354 \cs_new:Npn \__fp_parse_round_after:wN #1; #2
12355   {
12356     + #2 \exp_after:wN ;
12357     \__int_value:w \__int_eval:w #1 + \__fp_parse_exponent:N
12358   }
```

(*End 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 will expand to `+\c_zero` or `+\c_one`, 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 `+\c_zero` or `+\c_one` 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`.

```
12359 \cs_new:Npn \__fp_parse_small_round:NN #1#2
12360   {
12361     \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
12362       +
12363       \exp_after:wN \__fp_round_s:NNNw
12364       \exp_after:wN 0
12365       \exp_after:wN #1
12366       \exp_after:wN #2
12367       \__int_value:w \__int_eval:w
12368         \exp_after:wN \__fp_parse_round_after:wN
12369         \__int_value:w \__int_eval:w \c_zero * \__int_eval:w \c_zero
12370           \exp_after:wN \__fp_parse_round_loop:N
12371           \exp:w
12372     \else:
12373       \__fp_parse_exponent:Nw #2
12374     \fi:
12375     \__fp_parse_expand:w
12376   }
```

(*End definition for* `\__fp_parse_small_round:NN` *and* `\__fp_parse_round_after:wN`.)

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.

```
12377 \cs_new:Npn \__fp_parse_large_round:NN #1#2
12378   {
12379     \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
12380       +
12381       \exp_after:wN \__fp_round_s:NNNw
12382       \exp_after:wN 0
12383       \exp_after:wN #1
12384       \exp_after:wN #2
12385       \__int_value:w \__int_eval:w
12386         \exp_after:wN \__fp_parse_large_round_aux:wNN
12387         \__int_value:w \__int_eval:w \c_one
12388           \exp_after:wN \__fp_parse_round_loop:N
12389     \else: %^^A could be dot, or e, or other
12390       \exp_after:wN \__fp_parse_large_round_test:NN
12391       \exp_after:wN #1
12392       \exp_after:wN #2
12393     \fi:
12394   }
12395 \cs_new:Npn \__fp_parse_large_round_test:NN #1#2
12396   {
12397     \if:w . \exp_not:N #2
12398       \exp_after:wN \__fp_parse_small_round:NN
12399       \exp_after:wN #1
12400       \exp:w
12401     \else:
12402       \__fp_parse_exponent:Nw #2
12403     \fi:
12404     \__fp_parse_expand:w
12405   }
12406 \cs_new:Npn \__fp_parse_large_round_aux:wNN #1 ; #2 #3
12407   {
12408     + #2
12409     \exp_after:wN \__fp_parse_round_after:wN
12410     \__int_value:w \__int_eval:w #1
12411       \if:w . \exp_not:N #3
12412         + \c_zero * \__int_eval:w \c_zero
12413           \exp_after:wN \__fp_parse_round_loop:N
12414           \exp:w \exp_after:wN \__fp_parse_expand:w
12415       \else:
12416         \exp_after:wN ;
12417         \exp_after:wN \c_zero
12418         \exp_after:wN #3
12419       \fi:
12420   }
```

(*End definition for* `\__fp_parse_large_round:NN`, `\__fp_parse_large_round_test:NN`, *and* `\__fp_-`
`parse_large_round_aux:wNN`.)

25.4.5 Number: finding the exponent

Expansion is a little bit tricky here, in part because we accept input where multiplication
is implicit.

```
\@@_parse:n { 3.2 erf(0.1) }
\@@_parse:n { 3.2 e\l_my_int }
\@@_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}`
· · · ; 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 `\__int_eval:w` ... there if needed.

```
12421 \cs_new:Npn \__fp_parse_exponent:Nw #1 #2 \__fp_parse_expand:w
12422   {
12423     \exp_after:wN ;
12424     \__int_value:w #2 \__fp_parse_exponent:N #1
12425   }
```

(*End definition for* `\__fp_parse_exponent:Nw`.)

`\__fp_parse_exponent:N`   This function should be called within an `\__int_value:w` expansion (or within an integer
`\__fp_parse_exponent_aux:N`   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`, leave an exponent of 0. If there is an `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.

```
12426 \cs_new:Npn \__fp_parse_exponent:N #1
12427   {
12428     \if:w e \exp_not:N #1
12429       \exp_after:wN \__fp_parse_exponent_aux:N
12430       \exp:w
12431     \else:
12432       0 \__fp_parse_return_semicolon:w #1
```

```
12433          \fi:
12434          \__fp_parse_expand:w
12435      }
12436 \cs_new:Npn \__fp_parse_exponent_aux:N #1
12437      {
12438        \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #1
12439                        \c_zero \else: '#1 \fi: > '9 \exp_stop_f:
12440          0 \exp_after:wN ; \exp_after:wN e
12441        \else:
12442          \exp_after:wN \__fp_parse_exponent_sign:N
12443        \fi:
12444        #1
12445      }
```

(*End definition for* __fp_parse_exponent:N *and* __fp_parse_exponent_aux:N.)

__fp_parse_exponent_sign:N   Read signs one by one (if there is any).

```
12446 \cs_new:Npn \__fp_parse_exponent_sign:N #1
12447      {
12448        \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
12449          \exp_after:wN \__fp_parse_exponent_sign:N
12450          \exp:w \exp_after:wN \__fp_parse_expand:w
12451        \else:
12452          \exp_after:wN \__fp_parse_exponent_body:N
12453          \exp_after:wN #1
12454        \fi:
12455      }
```

(*End 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).

```
12456 \cs_new:Npn \__fp_parse_exponent_body:N #1
12457      {
12458        \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
12459          \token_to_str:N #1
12460          \exp_after:wN \__fp_parse_exponent_digits:N
12461          \exp:w
12462        \else:
12463          \__fp_parse_exponent_keep:NTF #1
12464            { \__fp_parse_return_semicolon:w #1 }
12465            {
12466              \exp_after:wN ;
12467              \exp:w
12468            }
12469        \fi:
12470        \__fp_parse_expand:w
12471      }
```

(*End 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.

```
12472 \cs_new:Npn \__fp_parse_exponent_digits:N #1
12473   {
12474     \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
12475       \token_to_str:N #1
12476       \exp_after:wN \__fp_parse_exponent_digits:N
12477       \exp:w
12478     \else:
12479       \__fp_parse_return_semicolon:w #1
12480     \fi:
12481     \__fp_parse_expand:w
12482   }
```

(End 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.

```
12483 \prg_new_conditional:Npnn \__fp_parse_exponent_keep:N #1 { TF }
12484   {
12485     \if_catcode:w \scan_stop: \exp_not:N #1
12486       \if_meaning:w \scan_stop: #1
12487         \if_int_compare:w
12488             \__str_if_eq_x:nn { \s__fp } { \exp_not:N #1 } = \c_zero
12489           0
12490           \__msg_kernel_expandable_error:nnn
12491             { kernel } { fp-after-e } { floating~point~ }
12492           \prg_return_true:
12493         \else:
12494           0
12495           \__msg_kernel_expandable_error:nnn
12496             { kernel } { bad-variable } {#1}
12497           \prg_return_false:
12498         \fi:
12499       \else:
12500         \if_int_compare:w
12501             \__str_if_eq_x:nn { \__int_value:w #1 } { \tex_the:D #1 }
12502           = \c_zero
12503           \__int_value:w #1
12504         \else:
12505           0
12506           \__msg_kernel_expandable_error:nnn
12507             { kernel } { fp-after-e } { dimension~#1 }
12508         \fi:
12509         \prg_return_false:
12510       \fi:
12511     \else:
12512       0
12513       \__msg_kernel_expandable_error:nnn
```

```
12514          { kernel } { fp-missing } { exponent }
12515        \prg_return_true:
12516      \fi:
12517    }
```

(*End definition for* `\__fp_parse_exponent_keep:NTF.`)

25.5 Constants, functions and prefix operators

25.5.1 Prefix operators

`\__fp_parse_prefix_+:Nw`   A unary `+` does nothing: we should continue looking for a number.

```
12518 \cs_new_eq:cN { __fp_parse_prefix_+:Nw } \__fp_parse_one:Nw
```

(*End definition for* `\__fp_parse_prefix_+:Nw.`)

`\_fp_parse_apply_unary: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`.

```
12519 \cs_new:Npn \__fp_parse_apply_unary:NNNwN #1#2#3#4@#5
12520    {
12521      #3 #2 #4 @
12522      \exp:w \exp_end_continue_f:w #5 #1
12523    }
```

(*End definition for* `\__fp_parse_apply_unary:NNNwN.`)

`\__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_twelve` of the
unary operator, then call the appropriate `\__fp_⟨operation⟩_o:w` function, where the
⟨*operation*⟩ is `set_sign` or `not`.

```
12524 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
12525    {
12526      \cs_new:cpn { __fp_parse_prefix_ #1 :Nw } ##1
12527        {
12528          \exp_after:wN \__fp_parse_apply_unary:NNNwN
12529          \exp_after:wN ##1
12530          \exp_after:wN #4
12531          \exp_after:wN #3
12532          \exp:w
12533          \if_int_compare:w #2 < ##1
12534            \__fp_parse_operand:Nw ##1
12535          \else:
12536            \__fp_parse_operand:Nw #2
12537          \fi:
12538          \__fp_parse_expand:w
12539        }
12540    }
12541 \__fp_tmp:w - \c_twelve \__fp_set_sign_o:w 2
12542 \__fp_tmp:w ! \c_twelve \__fp_not_o:w ?
```

(*End definition for* `\__fp_parse_prefix_-:Nw` *and* `\__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.

```
12543 \cs_new:cpn { __fp_parse_prefix_.:Nw } #1
12544   {
12545     \exp_after:wN \__fp_parse_infix_after_operand:NwN
12546     \exp_after:wN #1
12547     \exp:w \exp_end_continue_f:w
12548       \exp_after:wN \__fp_sanitize:wN
12549       \__int_value:w \__int_eval:w \c_zero \__fp_parse_strim_zeros:N
12550   }
```

(*End definition for* __fp_parse_prefix_.:Nw.)

The left parenthesis is treated as a unary prefix operator because it appears in exactly the same settings. Commas will be allowed if the previous precedence is 16 (function with multiple arguments). In this case, find an operand using the precedence 1; otherwise the precedence 0. 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 the array it found as an operand, fetching the following infix operator.

```
12551 \group_begin:
12552   \char_set_catcode_letter:N (
12553   \char_set_catcode_letter:N )
12554   \cs_new:Npn \__fp_parse_prefix_(:Nw #1
12555     {
12556       \exp_after:wN \__fp_parse_lparen_after:NwN
12557       \exp_after:wN #1
12558       \exp:w
12559       \if_int_compare:w #1 = \c_sixteen
12560         \__fp_parse_operand:Nw \c_one
12561       \else:
12562         \__fp_parse_operand:Nw \c_zero
12563       \fi:
12564       \__fp_parse_expand:w
12565     }
12566   \cs_new:Npn \__fp_parse_lparen_after:NwN #1#2 @ #3
12567     {
12568       \token_if_eq_meaning:NNTF #3 \__fp_parse_infix_):N
12569         {
12570           \__fp_exp_after_array_f:w #2 \s__fp_stop
12571           \exp_after:wN \__fp_parse_infix:NN
12572           \exp_after:wN #1
12573           \exp:w \__fp_parse_expand:w
12574         }
12575         {
12576           \__msg_kernel_expandable_error:nnn
12577             { kernel } { fp-missing } { ) }
12578           #2 @ \use_none:n #3
12579         }
12580     }
12581 \group_end:
```

(End definition for `\__fp_parse_prefix_(:Nw` *and* `__fp_parse_lparen_after:NwN.)*

`\@@_parse_prefix_,:Nw`
`\__fp_parse_prefix_):Nw`

The right parenthesis and the comma can appear as unary prefixes when an argument of a multi-argument function is missing (for instance the first and last ones in `max(,1,2,)` or in `rand()`). In single-argument functions (precedence 0 rather than 1) forbid this.

```
12582 \cs_new:cpn { __fp_parse_prefix_,:Nw } #1
12583   {
12584     \if_int_compare:w #1 = \c_one
12585       \exp_after:wN \__fp_parse_one:Nw
12586       \exp_after:wN #1
12587     \else:
12588       \__msg_kernel_expandable_error:nnn
12589         { kernel } { fp-missing-number } { , }
12590       \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
12591       \exp_after:wN \__fp_parse_infix:NN
12592       \exp_after:wN #1
12593       \exp_after:wN ,
12594     \fi:
12595   }
12596 \cs_new:cpn { __fp_parse_prefix_):Nw } #1
12597   {
12598     \if_int_compare:w #1 = \c_one
12599     \else:
12600       \__msg_kernel_expandable_error:nnn
12601         { kernel } { fp-missing-number } { ) }
12602       \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
12603     \fi:
12604     \__fp_parse_infix:NN #1 )
12605   }
```

(End definition for `\@@_parse_prefix_,:Nw` *and* `__fp_parse_prefix_):Nw.)*

25.5.2 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`.

```
12606 \cs_set_protected:Npn \__fp_tmp:w #1 #2
12607   {
12608     \cs_new:cpn { __fp_parse_word_#1:N }
12609       { \exp_after:wN #2 \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN }
12610   }
12611 \__fp_tmp:w { inf } \c_inf_fp
12612 \__fp_tmp:w { nan } \c_nan_fp
12613 \__fp_tmp:w { pi  } \c_pi_fp
12614 \__fp_tmp:w { deg } \c_one_degree_fp
12615 \__fp_tmp:w { true } \c_one_fp
12616 \__fp_tmp:w { false } \c_zero_fp
```

(End definition for `\__fp_parse_word_inf:N` *and others.)*

`\__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.

```
12617 \cs_set_protected:Npn \__fp_tmp:w #1 #2
12618   {
```

```
12619        \cs_new:cpn { __fp_parse_word_#1:N }
12620          {
12621            \__fp_exp_after_f:nw { \__fp_parse_infix:NN }
12622            \s__fp \__fp_chk:w 10 #2 ;
12623          }
12624      }
12625    \__fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
12626    \__fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
12627    \__fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
12628    \__fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
12629    \__fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
12630    \__fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
12631    \__fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
12632    \__fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
12633    \__fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
12634    \__fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
12635    \__fp_tmp:w {sp} { {-4} {1525} {8789} {0625} {0000} }
```

(*End 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`.

```
12636    \tl_map_inline:nn { {em} {ex} }
12637      {
12638        \cs_new:cpn { __fp_parse_word_#1:N }
12639          {
12640            \exp_after:wN \__fp_from_dim_test:ww
12641            \exp_after:wN 0 \exp_after:wN ,
12642            \__int_value:w \__dim_eval:w 1 #1 \exp_after:wN ;
12643            \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN
12644          }
12645      }
```

(*End definition for* `\__fp_parse_word_em:N` *and* `\__fp_parse_word_ex:N`.)

25.5.3 Functions

`\__fp_parse_unary_function:nNN`
`\__fp_parse_function:NNN`

```
12646    \cs_new:Npn \__fp_parse_unary_function:nNN #1#2#3
12647      {
12648        \exp_after:wN \__fp_parse_apply_unary:NNNwN
12649        \exp_after:wN #3
12650        \exp_after:wN #2
12651        \cs:w __fp_#1_o:w \exp_after:wN \cs_end:
12652        \exp:w
12653        \__fp_parse_operand:Nw \c_fifteen \__fp_parse_expand:w
12654      }
12655    \cs_new:Npn \__fp_parse_function:NNN #1#2#3
12656      {
12657        \exp_after:wN \__fp_parse_apply_unary:NNNwN
12658        \exp_after:wN #3
12659        \exp_after:wN #2
12660        \exp_after:wN #1
12661        \exp:w
```

```
12662          \__fp_parse_operand:Nw \c_sixteen \__fp_parse_expand:w
12663        }
```

(*End definition for* `\__fp_parse_unary_function:nNN` *and* `\__fp_parse_function:NNN`.)

`\__fp_parse_word_acot:N`
`\__fp_parse_word_acotd:N`
`\__fp_parse_word_atan:N`
`\__fp_parse_word_atand:N`
`\__fp_parse_word_max:N`
`\__fp_parse_word_min:N`
`\__fp_parse_word_rand:N`
`\__fp_parse_word_randint:N`

Those functions are also unary (not binary), but may receive a variable number of arguments. For `randint` we don't use the first argument `?`.

```
12664  \cs_new:Npn \__fp_parse_word_acot:N
12665    { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_i:nn }
12666  \cs_new:Npn \__fp_parse_word_acotd:N
12667    { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_ii:nn }
12668  \cs_new:Npn \__fp_parse_word_atan:N
12669    { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_i:nn }
12670  \cs_new:Npn \__fp_parse_word_atand:N
12671    { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_ii:nn }
12672  \cs_new:Npn \__fp_parse_word_max:N
12673    { \__fp_parse_function:NNN \__fp_minmax_o:Nw 2 }
12674  \cs_new:Npn \__fp_parse_word_min:N
12675    { \__fp_parse_function:NNN \__fp_minmax_o:Nw 0 }
12676  \cs_new:Npn \__fp_parse_word_rand:N
12677    { \__fp_parse_function:NNN \__fp_rand_o:Nw ? }
12678  \cs_new:Npn \__fp_parse_word_randint:N
12679    { \__fp_parse_function:NNN \__fp_randint_o:Nw ? }
```

(*End definition for* `\__fp_parse_word_acot:N` *and others.*)

`\__fp_parse_word_abs:N`
`\__fp_parse_word_exp:N`
`\__fp_parse_word_ln:N`
`\__fp_parse_word_sqrt:N`

Unary functions.

```
12680  \cs_new:Npn \__fp_parse_word_abs:N
12681    { \__fp_parse_unary_function:nNN { set_sign } 0 }
12682  \cs_new:Npn \__fp_parse_word_exp:N
12683    { \__fp_parse_unary_function:nNN {exp} ? }
12684  \cs_new:Npn \__fp_parse_word_ln:N
12685    { \__fp_parse_unary_function:nNN {ln} ? }
12686  \cs_new:Npn \__fp_parse_word_sqrt:N
12687    { \__fp_parse_unary_function:nNN {sqrt} ? }
```

(*End definition for* `\__fp_parse_word_abs:N` *and others.*)

`\__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.

```
12688  \tl_map_inline:nn
12689    {
12690      {acos} {acsc} {asec} {asin}
12691      {cos} {cot} {csc} {sec} {sin} {tan}
12692    }
12693    {
12694      \cs_new:cpn { __fp_parse_word_#1:N }
12695        { \__fp_parse_unary_function:nNN {#1} \use_i:nn }
12696      \cs_new:cpn { __fp_parse_word_#1d:N }
12697        { \__fp_parse_unary_function:nNN {#1} \use_ii:nn }
12698    }
```

(*End definition for* `\__fp_parse_word_acos:N` *and others.*)

```
12699 \cs_new:Npn \__fp_parse_word_trunc:N
12700    { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_zero:NNN }
12701 \cs_new:Npn \__fp_parse_word_floor:N
12702    { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_ninf:NNN }
12703 \cs_new:Npn \__fp_parse_word_ceil:N
12704    { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_pinf:NNN }
```

(*End definition for* __fp_parse_word_trunc:N, __fp_parse_word_floor:N, *and* __fp_parse_word_-
ceil:N.)

```
12705 \cs_new:Npn \__fp_parse_word_round:N #1#2
12706    {
12707      \if_meaning:w + #2
12708        \__fp_parse_round:Nw \__fp_round_to_pinf:NNN
12709      \else:
12710      \if_meaning:w 0 #2
12711        \__fp_parse_round:Nw \__fp_round_to_zero:NNN
12712      \else:
12713      \if_meaning:w - #2
12714        \__fp_parse_round:Nw \__fp_round_to_ninf:NNN
12715      \fi:
12716      \fi:
12717      \fi:
12718      \__fp_parse_function:NNN
12719        \__fp_round_o:Nw \__fp_round_to_nearest:NNN #1
12720      #2
12721    }
12722 \cs_new:Npn \__fp_parse_round:Nw
12723    #1 #2 \__fp_round_to_nearest:NNN #3#4 { #2 #1 #3 }
```

(*End definition for* __fp_parse_word_round:N *and* __fp_parse_round:Nw.)

25.6 Main functions

__fp_parse: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 will perform computations until reaching an operation with
precedence \c_minus_one or less, namely, the end of the expression. The marker \s__-
fp_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:.

```
12724 \cs_new:Npn \__fp_parse:n #1
12725    {
12726      \exp:w
12727        \exp_after:wN \__fp_parse_after:ww
12728      \exp:w
12729        \__fp_parse_operand:Nw \c_minus_one
12730        \__fp_parse_expand:w #1
12731        \s__fp_mark \__fp_parse_infix_end:N
12732      \s__fp_stop
12733    }
12734 \cs_new:Npn \__fp_parse_after:ww
12735    #1@ \__fp_parse_infix_end:N \s__fp_stop
```

```
12736        { \exp_end: #1 }
```

(*End definition for* __fp_parse:n *and* __fp_parse_after:ww.)

__fp_parse_o:n

```
12737 \cs_new:Npn \__fp_parse_o:n #1
12738   {
12739     \exp_after:wN \exp_after:wN
12740     \exp_after:wN \__fp_exp_after_o:w
12741       \__fp_parse:n {#1}
12742   }
```

(*End definition for* __fp_parse_o:n.)

__fp_parse_operand:Nw
__fp_parse_continue:NwN

The __fp_parse_operand This is just a shorthand which sets up both __fp_parse_-
continue and __fp_parse_one with the same precedence. Note the trailing \exp:w.
This function should be used with much care.

```
12743 \cs_new:Npn \__fp_parse_operand:Nw #1
12744   {
12745     \exp_end_continue_f:w
12746     \exp_after:wN \__fp_parse_continue:NwN
12747     \exp_after:wN #1
12748     \exp:w \exp_end_continue_f:w
12749     \exp_after:wN \__fp_parse_one:Nw
12750     \exp_after:wN #1
12751     \exp:w
12752   }
12753 \cs_new:Npn \__fp_parse_continue:NwN #1 #2 @ #3 { #3 #1 #2 @ }
```

(*End definition for* __fp_parse_operand:Nw *and* __fp_parse_continue:NwN.)

__fp_parse_apply_binary:NwNwN

Receives ⟨*precedence*⟩ ⟨*operand*$_1$⟩ @ ⟨*operation*⟩ ⟨*operand*$_2$⟩ @ ⟨*infix command*⟩. Builds the
appropriate call to the ⟨*operation*⟩ #3.

```
12754 \cs_new:Npn \__fp_parse_apply_binary:NwNwN #1 #2@ #3 #4@ #5
12755   {
12756     \exp_after:wN \__fp_parse_continue:NwN
12757     \exp_after:wN #1
12758     \exp:w \exp_end_continue_f:w \cs:w __fp_#3_o:ww \cs_end: #2 #4
12759     \exp:w \exp_end_continue_f:w #5 #1
12760   }
```

(*End definition for* __fp_parse_apply_binary:NwNwN.)

25.7 Infix operators

__fp_parse_infix_after_operand:NwN

```
12761 \cs_new:Npn \__fp_parse_infix_after_operand:NwN #1 #2;
12762   {
12763     \__fp_exp_after_f:nw { \__fp_parse_infix:NN #1 }
12764     #2;
12765   }
12766 \group_begin:
12767   \char_set_catcode_letter:N \*
12768   \cs_new:Npn \__fp_parse_infix:NN #1 #2
```

```
12769        {
12770          \if_catcode:w \scan_stop: \exp_not:N #2
12771            \if_int_compare:w
12772                \__str_if_eq_x:nn { \s__fp_mark } { \exp_not:N #2 }
12773                = \c_zero
12774              \exp_after:wN \exp_after:wN
12775              \exp_after:wN \__fp_parse_infix_mark:NNN
12776            \else:
12777              \exp_after:wN \exp_after:wN
12778              \exp_after:wN \__fp_parse_infix_juxtapose:N
12779            \fi:
12780          \else:
12781            \if_int_compare:w
12782                \__int_eval:w
12783                  ( '#2 \if_int_compare:w '#2 > 'Z - \c_thirty_two \fi: )
12784                  / 26
12785                = \c_three
12786              \exp_after:wN \exp_after:wN
12787              \exp_after:wN \__fp_parse_infix_juxtapose:N
12788            \else:
12789              \exp_after:wN \__fp_parse_infix_check:NNN
12790              \cs:w
12791                __fp_parse_infix_ \token_to_str:N #2 :N
12792                \exp_after:wN \exp_after:wN \exp_after:wN
12793              \cs_end:
12794            \fi:
12795          \fi:
12796          #1
12797          #2
12798        }
12799      \cs_new:Npn \__fp_parse_infix_check:NNN #1#2#3
12800        {
12801          \if_meaning:w \scan_stop: #1
12802            \__msg_kernel_expandable_error:nnn
12803              { kernel } { fp-missing } { * }
12804            \exp_after:wN \__fp_parse_infix_*:N
12805            \exp_after:wN #2
12806            \exp_after:wN #3
12807          \else:
12808            \exp_after:wN #1
12809            \exp_after:wN #2
12810            \exp:w \exp_after:wN \__fp_parse_expand:w
12811          \fi:
12812        }
12813    \group_end:
```

(*End definition for* `\__fp_parse_infix_after_operand:NwN.`)

25.7.1 Closing parentheses and commas

`\__fp_parse_infix_mark:NNN`   As an infix operator, `\s__fp_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.

```
12814    \cs_new:Npn \__fp_parse_infix_mark:NNN #1#2#3 { #3 #1 }
```

601

(*End 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.

```
12815 \cs_new:Npn \__fp_parse_infix_end:N #1
12816   { @ \use_none:n \__fp_parse_infix_end:N }
```

(*End 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.

```
12817 \group_begin:
12818   \char_set_catcode_letter:N \)
12819   \cs_new:Npn \__fp_parse_infix_):N #1
12820     {
12821       \if_int_compare:w #1 < \c_zero
12822         \__msg_kernel_expandable_error:nnn { kernel } { fp-extra } { ) }
12823       \exp_after:wN \__fp_parse_infix:NN
12824       \exp_after:wN #1
12825       \exp:w \exp_after:wN \__fp_parse_expand:w
12826     \else:
12827       \exp_after:wN @
12828       \exp_after:wN \use_none:n
12829       \exp_after:wN \__fp_parse_infix_):N
12830     \fi:
12831     }
12832 \group_end:
```

(*End definition for* __fp_parse_infix_):N.)

__fp_parse_infix_,:N

```
12833 \group_begin:
12834   \char_set_catcode_letter:N \,
12835   \cs_new:Npn \__fp_parse_infix_,:N #1
12836     {
12837       \if_int_compare:w #1 > \c_one
12838         \exp_after:wN @
12839         \exp_after:wN \use_none:n
12840         \exp_after:wN \__fp_parse_infix_,:N
12841       \else:
12842         \if_int_compare:w #1 = \c_one
12843           \exp_after:wN \__fp_parse_infix_comma:w
12844           \exp:w
12845         \else:
12846           \exp_after:wN \__fp_parse_infix_comma_gobble:w
12847           \exp:w
12848         \fi:
12849         \__fp_parse_operand:Nw \c_one
12850         \exp_after:wN \__fp_parse_expand:w
12851       \fi:
12852     }
12853 \cs_new:Npn \__fp_parse_infix_comma:w #1 @
12854   { #1 @ \use_none:n }
12855 \cs_new:Npn \__fp_parse_infix_comma_gobble:w #1 @
```

```
12856        {
12857          \__msg_kernel_expandable_error:nn { kernel } { fp-extra-comma }
12858          @ \use_none:n
12859        }
12860    \group_end:
```

(*End definition for* `\__fp_parse_infix_,:N.`)

25.7.2 Usual infix operators

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.

The odd requirement to set `\+` here is to cover the case where expl3 is loaded by plain TeX: `\+` is an `\outer` macro there, and so the following code would otherwise give an error in that case.

```
12861  \group_begin:
12862  ⟨*package⟩
12863    \cs_set:Npn \+ { }
12864  ⟨/package⟩
12865    \char_set_catcode_other:N \&
12866    \char_set_catcode_letter:N \^
12867    \char_set_catcode_letter:N \/
12868    \char_set_catcode_letter:N \-
12869    \char_set_catcode_letter:N \+
12870    \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
12871      {
12872        \cs_new:Npn #1 ##1
12873          {
12874            \if_int_compare:w ##1 < #3
12875              \exp_after:wN @
12876              \exp_after:wN \__fp_parse_apply_binary:NwNwN
12877              \exp_after:wN #2
12878              \exp:w
12879              \__fp_parse_operand:Nw #4
12880              \exp_after:wN \__fp_parse_expand:w
12881            \else:
12882              \exp_after:wN @
12883              \exp_after:wN \use_none:n
12884              \exp_after:wN #1
12885            \fi:
12886          }
12887      }
12888    \__fp_tmp:w \__fp_parse_infix_^:N    ^ \c_fifteen \c_fourteen
12889    \__fp_tmp:w \__fp_parse_infix_/:N    / \c_ten      \c_ten
12890    \__fp_tmp:w \__fp_parse_infix_mul:N * \c_ten      \c_ten
12891    \__fp_tmp:w \__fp_parse_infix_-:N    - \c_nine     \c_nine
12892    \__fp_tmp:w \__fp_parse_infix_+:N    + \c_nine     \c_nine
12893    \__fp_tmp:w \__fp_parse_infix_and:N & \c_five     \c_five
12894    \__fp_tmp:w \__fp_parse_infix_or:N  | \c_four     \c_four
12895  \group_end:
```

(*End definition for* `\__fp_parse_infix_+:N` *and others.*)

25.7.3 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_juxtapose:N`.

```
12896 \cs_new:cpn { __fp_parse_infix_(:N } #1
12897   { \__fp_parse_infix_juxtapose:N #1 ( }
```

(*End definition for* `\__fp_parse_infix_(:N.`)

`\_fp_parse_infix_juxtapose:N`
`\_fp_parse_apply_juxtapose:NwwN`

Juxtaposition follows the same scheme as other binary operations, but calls `\__-fp_parse_apply_juxtapose:NwwN` rather than directly calling `\__fp_parse_apply_-binary:NwNwN`. This lets us catch errors such as `...(1,2,3)pt` where one operand of the juxtaposition is not a single number: both `#3` and `#5` of the `apply` auxiliary must be empty.

```
12898 \cs_new:Npn \__fp_parse_infix_juxtapose:N #1
12899   {
12900     \if_int_compare:w #1 < \c_ten
12901       \exp_after:wN @
12902       \exp_after:wN \__fp_parse_apply_juxtapose:NwwN
12903       \exp:w
12904       \__fp_parse_operand:Nw \c_ten
12905       \exp_after:wN \__fp_parse_expand:w
12906     \else:
12907       \exp_after:wN @
12908       \exp_after:wN \use_none:n
12909       \exp_after:wN \__fp_parse_infix_juxtapose:N
12910     \fi:
12911   }
12912 \cs_new:Npn \__fp_parse_apply_juxtapose:NwwN #1 #2;#3@ #4;#5@
12913   {
12914     \if_catcode:w ^ \tl_to_str:n { #3 #5 } ^
12915     \else:
12916       \__fp_error:nffn { invalid-ii }
12917         { \__fp_array_to_clist:n { #2; #3 } }
12918         { \__fp_array_to_clist:n { #4; #5 } }
12919         { }
12920     \fi:
12921     \__fp_parse_apply_binary:NwNwN #1 #2;@ * #4;@
12922   }
```

(*End definition for* `\__fp_parse_infix_juxtapose:N` *and* `\__fp_parse_apply_juxtapose:NwwN.`)

25.7.4 Multi-character cases

`\__fp_parse_infix_*:N`

```
12923 \group_begin:
12924   \char_set_catcode_letter:N ^
12925   \cs_new:cpn { __fp_parse_infix_*:N } #1#2
12926     {
12927       \if:w * \exp_not:N #2
12928         \exp_after:wN \__fp_parse_infix_^:N
12929         \exp_after:wN #1
12930       \else:
```

```
12931          \exp_after:wN \__fp_parse_infix_mul:N
12932          \exp_after:wN #1
12933          \exp_after:wN #2
12934        \fi:
12935      }
12936  \group_end:
```

(End definition for `\__fp_parse_infix_*:N`*.)*

```
12937  \group_begin:
12938    \char_set_catcode_letter:N \|
12939    \char_set_catcode_letter:N \&
12940    \cs_new:Npn \__fp_parse_infix_|:N #1#2
12941      {
12942        \if:w | \exp_not:N #2
12943          \exp_after:wN \__fp_parse_infix_|:N
12944          \exp_after:wN #1
12945          \exp:w \exp_after:wN \__fp_parse_expand:w
12946        \else:
12947          \exp_after:wN \__fp_parse_infix_or:N
12948          \exp_after:wN #1
12949          \exp_after:wN #2
12950        \fi:
12951      }
12952    \cs_new:Npn \__fp_parse_infix_&:N #1#2
12953      {
12954        \if:w & \exp_not:N #2
12955          \exp_after:wN \__fp_parse_infix_&:N
12956          \exp_after:wN #1
12957          \exp:w \exp_after:wN \__fp_parse_expand:w
12958        \else:
12959          \exp_after:wN \__fp_parse_infix_and:N
12960          \exp_after:wN #1
12961          \exp_after:wN #2
12962        \fi:
12963      }
12964  \group_end:
```

(End definition for `\__fp_parse_infix_|:Nw` *and* `\__fp_parse_infix_&:Nw`*.)*

25.7.5 Ternary operator

```
12965  \group_begin:
12966    \char_set_catcode_letter:N \?
12967    \cs_new:Npn \__fp_parse_infix_?:N #1
12968      {
12969        \if_int_compare:w #1 < \c_three
12970          \exp_after:wN @
12971          \exp_after:wN \__fp_ternary:NwwN
12972          \exp:w
12973          \__fp_parse_operand:Nw \c_three
12974          \exp_after:wN \__fp_parse_expand:w
```

605

```
12975          \else:
12976            \exp_after:wN @
12977            \exp_after:wN \use_none:n
12978            \exp_after:wN \__fp_parse_infix_?:N
12979          \fi:
12980        }
12981      \cs_new:Npn \__fp_parse_infix_::N #1
12982        {
12983          \if_int_compare:w #1 < \c_three
12984            \__msg_kernel_expandable_error:nnnn
12985              { kernel } { fp-missing } { ? } { ~for~?: }
12986            \exp_after:wN @
12987            \exp_after:wN \__fp_ternary_auxii:NwwN
12988            \exp:w
12989            \__fp_parse_operand:Nw \c_two
12990            \exp_after:wN \__fp_parse_expand:w
12991          \else:
12992            \exp_after:wN @
12993            \exp_after:wN \use_none:n
12994            \exp_after:wN \__fp_parse_infix_::N
12995          \fi:
12996        }
12997    \group_end:
```

(*End definition for* `\__fp_parse_infix_?:N` *and* `\__fp_parse_infix_::N`.)

25.7.6 Comparisons

```
12998  \cs_new:cpn { __fp_parse_infix_<:N } #1
12999    {
13000      \__fp_parse_compare:NNNNNNN #1 \c_one
13001        \c_zero  \c_zero \c_zero \c_zero <
13002    }
13003  \cs_new:cpn { __fp_parse_infix_=:N } #1
13004    {
13005      \__fp_parse_compare:NNNNNNN #1 \c_one
13006        \c_zero \c_zero \c_zero \c_zero =
13007    }
13008  \cs_new:cpn { __fp_parse_infix_>:N } #1
13009    {
13010      \__fp_parse_compare:NNNNNNN #1 \c_one
13011        \c_zero \c_zero \c_zero \c_zero >
13012    }
13013  \cs_new:cpn { __fp_parse_infix_!:N } #1
13014    {
13015      \exp_after:wN \__fp_parse_compare:NNNNNNN
13016      \exp_after:wN #1
13017      \exp_after:wN \c_zero
13018      \exp_after:wN \c_one
13019      \exp_after:wN \c_one
13020      \exp_after:wN \c_one
13021      \exp_after:wN \c_one
13022    }
```

606

```
13023  \cs_new:Npn \__fp_parse_excl_error:
13024    {
13025      \__msg_kernel_expandable_error:nnnn
13026        { kernel } { fp-missing } { = } { ~after~!. }
13027    }
13028  \cs_new:Npn \__fp_parse_compare:NNNNNNN #1
13029    {
13030      \if_int_compare:w #1 < \c_seven
13031        \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
13032        \exp_after:wN \__fp_parse_excl_error:
13033      \else:
13034        \exp_after:wN @
13035        \exp_after:wN \use_none:n
13036        \exp_after:wN \__fp_parse_compare:NNNNNNN
13037      \fi:
13038    }
13039  \cs_new:Npn \__fp_parse_compare_auxi:NNNNNNN #1#2#3#4#5#6#7
13040    {
13041      \if_case:w
13042            \if_catcode:w \scan_stop: \exp_not:N #7
13043              \c_minus_one
13044            \else:
13045              \__int_eval:w '#7 - '< \__int_eval_end:
13046            \fi:
13047          \__fp_parse_compare_auxii:NNNNN #2#2#4#5#6
13048      \or: \__fp_parse_compare_auxii:NNNNN #2#3#2#5#6
13049      \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#2#6
13050      \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#5#2
13051      \else: #1 \__fp_parse_compare_end:NNNNw #3#4#5#6#7
13052      \fi:
13053    }
13054  \cs_new:Npn \__fp_parse_compare_auxii:NNNNN #1#2#3#4#5
13055    {
13056      \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
13057      \exp_after:wN \prg_do_nothing:
13058      \exp_after:wN #1
13059      \exp_after:wN #2
13060      \exp_after:wN #3
13061      \exp_after:wN #4
13062      \exp_after:wN #5
13063      \exp:w \exp_after:wN \__fp_parse_expand:w
13064    }
13065  \cs_new:Npn \__fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
13066    {
13067      \fi:
13068      \exp_after:wN @
13069      \exp_after:wN \__fp_parse_apply_compare:NwNNNNNwN
13070      \exp_after:wN \c_one_fp
13071      \exp_after:wN #1
13072      \exp_after:wN #2
13073      \exp_after:wN #3
13074      \exp_after:wN #4
13075      \exp:w
13076      \__fp_parse_operand:Nw \c_seven \__fp_parse_expand:w #5
```

```
13077      }
13078 \cs_new:Npn \__fp_parse_apply_compare:NwNNNNNwN
13079      #1 #2@ #3 #4#5#6#7 #8@ #9
13080      {
13081        \if_int_odd:w
13082          \if_meaning:w \c_zero_fp #3
13083            \c_zero
13084          \else:
13085            \if_case:w \__fp_compare_back:ww #8 #2 \exp_stop_f:
13086              #5 \or: #6 \or: #7 \else: #4
13087            \fi:
13088          \fi:
13089          \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
13090          \exp_after:wN \c_one_fp
13091        \else:
13092          \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
13093          \exp_after:wN \c_zero_fp
13094        \fi:
13095        #1 #8 #9
13096      }
13097 \cs_new:Npn \__fp_parse_apply_compare_aux:NNwN #1 #2 #3; #4
13098      {
13099        \if_meaning:w \__fp_parse_compare:NNNNNNN #4
13100          \exp_after:wN \__fp_parse_continue_compare:NNwNN
13101          \exp_after:wN #1
13102          \exp_after:wN #2
13103          \exp:w \exp_end_continue_f:w
13104          \__fp_exp_after_o:w #3;
13105          \exp:w \exp_end_continue_f:w
13106        \else:
13107          \exp_after:wN \__fp_parse_continue:NwN
13108          \exp_after:wN #2
13109          \exp:w \exp_end_continue_f:w
13110          \exp_after:wN #1
13111          \exp:w \exp_end_continue_f:w
13112        \fi:
13113        #4 #2
13114      }
13115 \cs_new:Npn \__fp_parse_continue_compare:NNwNN #1#2 #3@ #4#5
13116      { #4 #2 #3@ #1 }
```

(*End definition for* `\__fp_parse_infix_<:N` *and others.*)

25.8   Candidate: defining new l3fp functions

`\fp_function:Nw`   Parse the argument of the function #1 using `\__fp_parse_operand:Nw` with a precedence of 16, and pass the function and argument to `\__fp_function_apply:nw`.

```
13117 \cs_new:Npn \fp_function:Nw #1
13118      {
13119        \exp_after:wN \__fp_function_apply:nw
13120        \exp_after:wN #1
13121        \exp:w
13122          \__fp_parse_operand:Nw \c_sixteen \__fp_parse_expand:w
13123      }
```

`\fp_new_function:Npn`
`\__fp_new_function:NNnnn`
`\__fp_new_function:Ncfnn`
`\__fp_function_args:Nwn`

Save the code provided by the user in the control sequence `\__fp_user_#1`. Define `#1` to call `\__fp_function_apply:nw` after parsing one operand using `\__fp_parse_-operand:Nw` with precedence 16. The auxiliary `\__fp_function_args:Nwn` receives the user function and the number of arguments (half of the number of tokens in the parameter text `#2`), followed by the operand (as a token list of floating points). It checks the number of arguments, and applies the user function to the arguments (without the outer brace group).

```
13124 \cs_new_protected:Npn \fp_new_function:Npn #1#2#
13125   {
13126     \__fp_new_function:Ncfnn #1
13127       { __fp_user_ \cs_to_str:N #1 }
13128       { \int_eval:n { \tl_count:n {#2} / \c_two } }
13129       {#2}
13130   }
13131 \cs_new_protected:Npn \__fp_new_function:NNnnn #1#2#3#4#5
13132   {
13133     \cs_new:Npn #1
13134       {
13135         \exp_after:wN \__fp_function_apply:nw \exp_after:wN
13136           {
13137             \exp_after:wN \__fp_function_args:Nwn
13138             \exp_after:wN #2
13139             \__int_value:w #3 \exp_after:wN ; \exp_after:wN
13140           }
13141         \exp:w
13142           \__fp_parse_operand:Nw \c_sixteen \__fp_parse_expand:w
13143       }
13144     \cs_new:Npn #2 #4 {#5}
13145   }
13146 \cs_generate_variant:Nn \__fp_new_function:NNnnn { Ncf }
13147 \cs_new:Npn \__fp_function_args:Nwn #1#2; #3
13148   {
13149     \int_compare:nNnTF { \tl_count:n {#3} } = {#2}
13150       { #1 #3 }
13151       {
13152         \__msg_kernel_expandable_error:nnnnn
13153           { kernel } { fp-num-args } { #1() } {#2} {#2}
13154         \c_nan_fp
13155       }
13156   }
```

`\__fp_function_apply:nw`
`\__fp_function_store:wwNwnn`
`\__fp_function_store_end:wnnn`

The auxiliary `\__fp_function_apply:nw` is called after parsing an operand, so it receives some code `#1`, then the operand ending with `@`, then a function such as `\__fp_parse_-infix_+:N` (but not always of this form, see comparisons for instance). Package the operand (an array) into a token list with floating point items: this is the role of `\__fp_-function_store:wwNwnn` and `\__fp_function_store_end:wnnn`. Then apply `\__fp_-parse:n` to the code `#1` followed by a brace group with this token list. This results in a floating point result, which will correctly be parsed as the next operand of whatever was

looking for one. The trailing `\s__fp_mark` is used as a special infix operator to indicate that the next token has already gone through `\__fp_parse_infix:NN`.

```
13157 \cs_new:Npn \__fp_function_apply:nw #1#2 @
13158   {
13159     \__fp_parse:n
13160       {
13161         \__fp_function_store:wwNwnn #2
13162           \s__fp_mark \__fp_function_store:wwNwnn ;
13163           \s__fp_mark \__fp_function_store_end:wnnn
13164         \s__fp_stop { } { } {#1}
13165       }
13166     \s__fp_mark
13167   }
13168 \cs_new:Npn \__fp_function_store:wwNwnn
13169     #1; #2 \s__fp_mark #3#4 \s__fp_stop #5#6
13170   { #3 #2 \s__fp_mark #3#4 \s__fp_stop { #5 #6 } { { #1; } } }
13171 \cs_new:Npn \__fp_function_store_end:wnnn
13172     #1 \s__fp_stop #2#3#4
13173   { #4 {#2} }
```

(*End definition for* `\__fp_function_apply:nw`, `\__fp_function_store:wwNwnn`, *and* `\__fp_function_-store_end:wnnn`.)

25.9 Messages

```
13174 \__msg_kernel_new:nnn { kernel } { unknown-fp-word }
13175   { Unknown~fp~word~#1. }
13176 \__msg_kernel_new:nnn { kernel } { fp-missing }
13177   { Missing~#1~inserted #2. }
13178 \__msg_kernel_new:nnn { kernel } { fp-extra }
13179   { Extra~#1~ignored. }
13180 \__msg_kernel_new:nnn { kernel } { fp-early-end }
13181   { Premature~end~in~fp~expression. }
13182 \__msg_kernel_new:nnn { kernel } { fp-after-e }
13183   { Cannot~use~#1 after~'e'. }
13184 \__msg_kernel_new:nnn { kernel } { fp-missing-number }
13185   { Missing~number~before~'#1'. }
13186 \__msg_kernel_new:nnn { kernel } { fp-unknown-symbol }
13187   { Unknown~symbol~#1~ignored. }
13188 \__msg_kernel_new:nnn { kernel } { fp-extra-comma }
13189   { Unexpected~comma:~extra~arguments~ignored. }
13190 \__msg_kernel_new:nnn { kernel } { fp-num-args }
13191   { #1~expects~between~#2~and~#3~arguments. }
13192 ⟨*package⟩
13193 \cs_if_exist:cT { @unexpandable@protect }
13194   {
13195     \__msg_kernel_new:nnn { kernel } { fp-robust-cmd }
13196       { Robust~command~#1 invalid~in~fp~expression! }
13197   }
13198 ⟨/package⟩
13199 ⟨/initex | package⟩
```

26 l3fp-logic Implementation

13201 ⟨@@=fp⟩

26.1   Syntax of internal functions

- __fp_compare_npos:nwnw {⟨expo₁⟩} ⟨body₁⟩ ; {⟨expo₂⟩} ⟨body₂⟩ ;

- __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.

26.2   Existence test

\fp_if_exist_p:N
\fp_if_exist_p:c
\fp_if_exist:N*TF*
\fp_if_exist:c*TF*

Copies of the cs functions defined in l3basics.

```
13202 \prg_new_eq_conditional:NNn \fp_if_exist:N \cs_if_exist:N { TF , T , F , p }
13203 \prg_new_eq_conditional:NNn \fp_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(*End definition for* \fp_if_exist:NTF*. This function is documented on page 183.*)

26.3   Comparison

\fp_compare_p:n
\fp_compare:n*TF*

__fp_compare_return:w

Within floating point expressions, comparison operators are treated as operations, so we evaluate #1, then compare with 0.

```
13204 \prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }
13205   {
13206     \exp_after:wN \__fp_compare_return:w
13207     \exp:w \exp_end_continue_f:w \__fp_parse:n {#1}
13208   }
13209 \cs_new:Npn \__fp_compare_return:w \s__fp \__fp_chk:w #1#2;
13210   {
13211     \if_meaning:w 0 #1
13212       \prg_return_false:
13213     \else:
13214       \prg_return_true:
13215     \fi:
13216   }
```

(*End definition for* \fp_compare:nTF *and* __fp_compare_return:w*. These functions are documented on page 184.*)

\fp_compare_p:nNn
\fp_compare:nNn*TF*
__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:ww, defined below. Compare the result with '#2-'=, which is −1 for <, 0 for =, 1 for > and 2 for ?.

```
13217 \prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
13218   {
13219     \if_int_compare:w
13220       \exp_after:wN \__fp_compare_aux:wn
13221         \exp:w \exp_end_continue_f:w \__fp_parse:n {#1} {#3}
```

611

```
13222              = \__int_eval:w `#2 - `= \__int_eval_end:
13223            \prg_return_true:
13224          \else:
13225            \prg_return_false:
13226          \fi:
13227       }
13228     \cs_new:Npn \__fp_compare_aux:wn #1; #2
13229       {
13230         \exp_after:wN \__fp_compare_back:ww
13231           \exp:w \exp_end_continue_f:w \__fp_parse:n {#2} #1;
13232       }
```

(*End definition for* \fp_compare:nNnTF *and* __fp_compare_aux:wn. *These functions are documented on page* *184*.)

__fp_compare_back:ww   __fp_compare_back:ww ⟨y⟩ ; ⟨x⟩ ;
__fp_compare_nan:w   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.

```
13233     \cs_new:Npn \__fp_compare_back:ww
13234         \s__fp \__fp_chk:w #1 #2 #3;
13235         \s__fp \__fp_chk:w #4 #5 #6;
13236       {
13237         \__int_value:w
13238           \if_meaning:w 3 #1 \exp_after:wN \__fp_compare_nan:w \fi:
13239           \if_meaning:w 3 #4 \exp_after:wN \__fp_compare_nan:w \fi:
13240           \if_meaning:w 2 #5 - \fi:
13241           \if_meaning:w #2 #5
13242             \if_meaning:w #1 #4
13243               \if_meaning:w 1 #1
13244                 \__fp_compare_npos:nwnw #6; #3;
13245               \else:
13246                 0
13247               \fi:
13248             \else:
13249               \if_int_compare:w #4 < #1 - \fi: 1
13250             \fi:
13251           \else:
13252             \if_int_compare:w #1#4 = \c_zero
13253               0
13254             \else:
13255               1
13256             \fi:
13257           \fi:
13258         \exp_stop_f:
13259       }
13260     \cs_new:Npn \__fp_compare_nan:w #1 \exp_stop_f: { \c_two }
```

(*End definition for* __fp_compare_back:ww *and* __fp_compare_nan:w.)

__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.

```
13261 \cs_new:Npn \__fp_compare_npos:nwnw #1#2; #3#4;
13262   {
13263     \if_int_compare:w #1 = #3 \exp_stop_f:
13264       \__fp_compare_significand:nnnnnnnn #2 #4
13265     \else:
13266       \if_int_compare:w #1 < #3 - \fi: 1
13267     \fi:
13268   }
13269 \cs_new:Npn \__fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
13270   {
13271     \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
13272       \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
13273         0
13274       \else:
13275         \if_int_compare:w #3#4 < #7#8 - \fi: 1
13276       \fi:
13277     \else:
13278       \if_int_compare:w #1#2 < #5#6 - \fi: 1
13279     \fi:
13280   }
```

(*End definition for* __fp_compare_npos:nwnw *and* __fp_compare_significand:nnnnnnnn.)

26.4 Floating point expression loops

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.

```
13281 \cs_new:Npn \fp_do_until:nn #1#2
13282   {
13283     #2
13284     \fp_compare:nF {#1}
13285       { \fp_do_until:nn {#1} {#2} }
13286   }
13287 \cs_new:Npn \fp_do_while:nn #1#2
13288   {
13289     #2
13290     \fp_compare:nT {#1}
13291       { \fp_do_while:nn {#1} {#2} }
13292   }
13293 \cs_new:Npn \fp_until_do:nn #1#2
13294   {
13295     \fp_compare:nF {#1}
13296       {
13297         #2
13298         \fp_until_do:nn {#1} {#2}
13299       }
```

```
13300      }
13301 \cs_new:Npn \fp_while_do:nn #1#2
13302    {
13303      \fp_compare:nT {#1}
13304        {
13305          #2
13306          \fp_while_do:nn {#1} {#2}
13307        }
13308    }
```

(*End definition for* \fp_do_until:nn *and others. These functions are documented on page 185.*)

\fp_do_until:nNnn   As above but not using the nNn syntax.
\fp_do_while:nNnn
\fp_until_do:nNnn
\fp_while_do:nNnn

```
13309 \cs_new:Npn \fp_do_until:nNnn #1#2#3#4
13310    {
13311      #4
13312      \fp_compare:nNnF {#1} #2 {#3}
13313        { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
13314    }
13315 \cs_new:Npn \fp_do_while:nNnn #1#2#3#4
13316    {
13317      #4
13318      \fp_compare:nNnT {#1} #2 {#3}
13319        { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
13320    }
13321 \cs_new:Npn \fp_until_do:nNnn #1#2#3#4
13322    {
13323      \fp_compare:nNnF {#1} #2 {#3}
13324        {
13325          #4
13326          \fp_until_do:nNnn {#1} #2 {#3} {#4}
13327        }
13328    }
13329 \cs_new:Npn \fp_while_do:nNnn #1#2#3#4
13330    {
13331      \fp_compare:nNnT {#1} #2 {#3}
13332        {
13333          #4
13334          \fp_while_do:nNnn {#1} #2 {#3} {#4}
13335        }
13336    }
```

(*End definition for* \fp_do_until:nNnn *and others. These functions are documented on page 185.*)

\fp_step_function:nnnN   The approach here is much the same as \int_step_function:nnnN except we do not
\fp_step_function:nnnc   have a convenient low-level way to evaluation each argument to give a pure number.
__fp_step:wwwN   Instead, the internal parser is used to leave values in the internal format (and avoid a
__fp_step:NnnnN   more expensive f-type expansion of multiple uses of fp_eval:n).
__fp_step:NfnnN

```
13337 \cs_new:Npn \fp_step_function:nnnN #1#2#3
13338    {
13339      \exp_after:wN \__fp_step:wwwN
13340        \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#1} :
13341        \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#2} :
13342        \exp:w \exp_end_continue_f:w \__fp_parse:n {#3} :
```

614

```
13343     }
13344 \cs_generate_variant:Nn \fp_step_function:nnnN { nnnc }
```
Here, #1 will be passed to the output on the first pass of the loop. To avoid this being in internal form, an f-type expansion is used here on that value only. The second and third arguments will be converted during the loop itself.
```
13345 \cs_new:Npn \__fp_step:wwwN #1 : #2 : #3 : #4
13346   {
13347     \fp_compare:nNnTF {#2} > 0
13348       { \__fp_step:NfnnN > }
13349       {
13350         \fp_compare:nNnTF {#2} = 0
13351           {
13352             \__msg_kernel_expandable_error:nnn { kernel } { zero-step } {#4}
13353             \use_none:nnnn
13354           }
13355           { \__fp_step:NfnnN < }
13356       }
13357       { \fp_eval:n {#1} } {#2} {#3} #4
13358   }
13359
13360 \cs_new:Npn \__fp_step:NnnnN #1#2#3#4#5
13361   {
13362     \fp_compare:nNnF {#2} #1 {#4}
13363       {
13364         #5 {#2}
13365         \__fp_step:NfnnN
13366           #1 { \fp_eval:n { #2 + #3 } } {#3} {#4} #5
13367       }
13368   }
13369 \cs_generate_variant:Nn \__fp_step:NnnnN { Nf }
```
(*End definition for* \fp_step_function:nnnN *,* __fp_step:wwwN *, and* __fp_step:NnnnN*. These functions are documented on page 186.*)

\fp_step_inline:nnnn As for \int_step_inline:nnnn, create a global function and apply it, following up with a break point.
```
13370 \cs_new_protected:Npn \fp_step_inline:nnnn #1#2#3#4
13371   {
13372     \int_gincr:N \g__prg_map_int
13373     \cs_gset_protected:cpn { __prg_map_ \int_use:N \g__prg_map_int :w }
13374       ##1 {#4}
13375     \fp_step_function:nnnc {#1} {#2} {#3}
13376       { __prg_map_ \int_use:N \g__prg_map_int :w }
13377     \__prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__prg_map_int }
13378   }
```
(*End definition for* \fp_step_inline:nnnn*. This function is documented on page 186.*)

26.5 Extrema

__fp_minmax_o:Nw 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

615

kept. We append $-\infty$ (∞), for the case of an empty array, currently impossible. Since no number is smaller (larger) than that, it will never alter the maximum (minimum). The weird fp-like trailing marker breaks the loop correctly: see the precise definition of `\__fp_minmax_loop:Nww`.

```
13379 \cs_new:Npn \__fp_minmax_o:Nw #1#2 @
13380   {
13381     \if_meaning:w 0 #1
13382       \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN \c_one
13383     \else:
13384       \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN \c_minus_one
13385     \fi:
13386     #2
13387     \s__fp \__fp_chk:w 2 #1 \s__fp_exact ;
13388     \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;
13389   }
```

(*End definition for* `\__fp_minmax_o:Nw`.)

`\__fp_minmax_loop:Nww` The first argument is -1 or 1 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.

```
13390 \cs_new:Npn \__fp_minmax_loop:Nww
13391     #1 \s__fp \__fp_chk:w #2#3; \s__fp \__fp_chk:w #4#5;
13392   {
13393     \if_meaning:w 3 #4
13394       \if_meaning:w 3 #2
13395         \__fp_minmax_auxi:ww
13396       \else:
13397         \__fp_minmax_auxii:ww
13398       \fi:
13399     \else:
13400       \if_int_compare:w
13401           \__fp_compare_back:ww
13402             \s__fp \__fp_chk:w #4#5;
13403             \s__fp \__fp_chk:w #2#3;
13404         = #1
13405         \__fp_minmax_auxii:ww
13406       \else:
13407         \__fp_minmax_auxi:ww
13408       \fi:
13409     \fi:
13410     \__fp_minmax_loop:Nww #1
13411       \s__fp \__fp_chk:w #2#3;
13412       \s__fp \__fp_chk:w #4#5;
13413   }
```

(*End definition for* `\__fp_minmax_loop:Nww`.)

`\__fp_minmax_auxi:ww`
`\__fp_minmax_auxii:ww`

Keep the first/second number, and remove the other.

```
13414 \cs_new:Npn \__fp_minmax_auxi:ww  #1 \fi: \fi: #2 \s__fp #3 ; \s__fp #4;
13415   { \fi: \fi: #2 \s__fp #3 ; }
13416 \cs_new:Npn \__fp_minmax_auxii:ww #1 \fi: \fi: #2 \s__fp #3 ;
13417   { \fi: \fi: #2 }
```

(*End 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.

```
13418 \cs_new:Npn \__fp_minmax_break_o:w #1 \fi: \fi: #2 \s__fp #3; #4;
13419   { \fi: \__fp_exp_after_o:w \s__fp #3; }
```

(*End definition for* __fp_minmax_break_o:w.)

26.6   Boolean operations

__fp_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.

```
13420 \cs_new:cpn { __fp_not_o:w } #1 \s__fp \__fp_chk:w #2#3; @
13421   {
13422     \if_meaning:w 0 #2
13423       \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
13424     \else:
13425       \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
13426     \fi:
13427   }
```

(*End definition for* __fp_not_o:w.)

__fp_&_o:ww   For and, if the first number is zero, return it (with the same sign). Otherwise, return
__fp_|_o:ww   the second one. For or, the logic is reversed: if the first number is non-zero, return
__fp_and_return:wNw   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.

```
13428 \group_begin:
13429   \char_set_catcode_letter:N &
13430   \char_set_catcode_letter:N |
13431   \cs_new:Npn \__fp_&_o:ww #1 \s__fp \__fp_chk:w #2#3;
13432     {
13433       \if_meaning:w 0 #2 #1
13434         \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
13435       \fi:
13436       \__fp_exp_after_o:w
13437     }
13438   \cs_new:Npn \__fp_|_o:ww { \__fp_&_o:ww \else: }
13439 \group_end:
13440 \cs_new:Npn \__fp_and_return:wNw #1; \fi: #2#3; { \fi: #2 #1; }
```

(*End definition for* __fp_&_o:ww, __fp_|_o:ww, *and* __fp_and_return:wNw.)

26.7 Ternary operator

__fp_ternary:NwwN
__fp_ternary_auxi:NwwN
__fp_ternary_auxii:NwwN
__fp_ternary_loop_break:w
__fp_ternary_loop:Nw
__fp_ternary_map_break:
__fp_ternary_break_point:n

The first function receives the test and the true branch of the ?: ternary operator. It returns the true branch, unless the test branch is zero. In that case, the function returns a very specific nan. The second function receives the output of the first function, and the false branch. It returns the previous input, unless that is the special nan, in which case we return the false branch.

```
13441 \cs_new:Npn \__fp_ternary:NwwN #1 #2@ #3@ #4
13442   {
13443     \if_meaning:w \__fp_parse_infix_::N #4
13444       \__fp_ternary_loop:Nw
13445         #2
13446         \s__fp \__fp_chk:w { \__fp_ternary_loop_break:w } ;
13447       \__fp_ternary_break_point:n { \exp_after:wN \__fp_ternary_auxi:NwwN }
13448       \exp_after:wN #1
13449       \exp:w \exp_end_continue_f:w
13450       \__fp_exp_after_array_f:w #3 \s__fp_stop
13451       \exp_after:wN @
13452       \exp:w
13453         \__fp_parse_operand:Nw \c_two
13454         \__fp_parse_expand:w
13455     \else:
13456       \__msg_kernel_expandable_error:nnnn
13457         { kernel } { fp-missing } { : } { ~for~?: }
13458       \exp_after:wN \__fp_parse_continue:NwN
13459       \exp_after:wN #1
13460       \exp:w \exp_end_continue_f:w
13461       \__fp_exp_after_array_f:w #3 \s__fp_stop
13462       \exp_after:wN #4
13463       \exp_after:wN #1
13464     \fi:
13465   }
13466 \cs_new:Npn \__fp_ternary_loop_break:w
13467     #1 \fi: #2 \__fp_ternary_break_point:n #3
13468   {
13469     \c_zero = \c_zero \fi:
13470     \exp_after:wN \__fp_ternary_auxii:NwwN
13471   }
13472 \cs_new:Npn \__fp_ternary_loop:Nw \s__fp \__fp_chk:w #1#2;
13473   {
13474     \if_int_compare:w #1 > \c_zero
13475       \exp_after:wN \__fp_ternary_map_break:
13476     \fi:
13477     \__fp_ternary_loop:Nw
13478   }
13479 \cs_new:Npn \__fp_ternary_map_break: #1 \__fp_ternary_break_point:n #2 {#2}
13480 \cs_new:Npn \__fp_ternary_auxi:NwwN #1#2@#3@#4
13481   {
13482     \exp_after:wN \__fp_parse_continue:NwN
13483     \exp_after:wN #1
13484     \exp:w \exp_end_continue_f:w
13485     \__fp_exp_after_array_f:w #2 \s__fp_stop
13486     #4 #1
13487   }
```

```
13488  \cs_new:Npn \__fp_ternary_auxii:NwwN #1#2@#3@#4
13489    {
13490      \exp_after:wN \__fp_parse_continue:NwN
13491      \exp_after:wN #1
13492      \exp:w \exp_end_continue_f:w
13493      \__fp_exp_after_array_f:w #3 \s__fp_stop
13494      #4 #1
13495    }
```

(*End definition for* __fp_ternary:NwwN *and others.*)

```
13496  ⟨/initex | package⟩
```

27 **l3fp-basics** Implementation

```
13497  ⟨*initex | package⟩
```

```
13498  ⟨@@=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`.

27.1 Common to several operations

__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}.

```
13499  \cs_new:Npn \__fp_basics_pack_low:NNNNNw #1 #2#3#4#5 #6;
13500    { + #1 - \c_one ; {#2#3#4#5} {#6} ; }
13501  \cs_new:Npn \__fp_basics_pack_high:NNNNNw #1 #2#3#4#5 #6;
13502    {
13503      \if_meaning:w 2 #1
13504        \__fp_basics_pack_high_carry:w
13505      \fi:
13506      ; {#2#3#4#5} {#6}
13507    }
13508  \cs_new:Npn \__fp_basics_pack_high_carry:w \fi: ; #1
13509    { \fi: + \c_one ; {1000} }
```

(*End 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

I don't fully understand those functions, used for additions and divisions. Hence the name.

```
13510  \cs_new:Npn \__fp_basics_pack_weird_low:NNNNw #1 #2#3#4 #5;
13511    {
13512      \if_meaning:w 2 #1
```

619

```
13513        + \c_one
13514      \fi:
13515      \__int_eval_end:
13516      #2#3#4; {#5} ;
13517    }
13518 \cs_new:Npn \__fp_basics_pack_weird_high:NNNNNNNNw
13519    1 #1#2#3#4 #5#6#7#8 #9; { ; {#1#2#3#4} {#5#6#7#8} {#9} }
```

(*End definition for* __fp_basics_pack_weird_low:NNNNw *and* __fp_basics_pack_weird_high:NNNNNNNNw.)

27.2 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.

27.2.1 Sign, exponent, and special numbers

__fp_-_o:ww   A previous version of this function grabbed its two operands, changed the sign of the second, and called __fp_+_o:ww. However, for efficiency reasons, the operands were swapped in the process, which means that error messages ended up wrong. Now, 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 check that it was followed by \s__fp and not arbitrary junk.

```
13520 \cs_new:cpx { __fp_-_o:ww } \s__fp
13521    {
13522      \exp_not:c { __fp_+_o:ww }
13523      \exp_not:n { \s__fp \__fp_neg_sign:N }
13524    }
```

(*End 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 (equivalent to changing the ⟨*sign₂*⟩ of the second operand). 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 ⟨*sign₂*⟩ (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 ⟨*type₁*⟩ is greater than ⟨*type₂*⟩. Also note that we don't need to worry about ⟨*sign₂*⟩ in that case since the second operand is discarded.

```
13525 \cs_new:cpn { __fp_+_o:ww }
13526     \s__fp #1 \__fp_chk:w #2 #3 ; \s__fp \__fp_chk:w #4 #5
13527   {
13528     \if_case:w
13529       \if_meaning:w #2 #4
13530         #2 \exp_stop_f:
13531       \else:
13532         \if_int_compare:w #2 > #4 \exp_stop_f:
13533           \c_three
13534         \else:
13535           \c_minus_one
13536         \fi:
13537       \fi:
13538           \exp_after:wN \__fp_add_zeros_o:Nww \__int_value:w
13539     \or:   \exp_after:wN \__fp_add_normal_o:Nww \__int_value:w
13540     \or:   \exp_after:wN \__fp_add_inf_o:Nww \__int_value:w
13541     \or:   \__fp_case_return_i_o:ww
13542     \else: \exp_after:wN \__fp_add_return_ii_o:Nww \__int_value:w
13543     \fi:
13544     #1 #5
13545     \s__fp \__fp_chk:w #2 #3 ;
13546     \s__fp \__fp_chk:w #4 #5
13547   }
```

(*End 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.

```
13548 \cs_new:Npn \__fp_add_return_ii_o:Nww #1 #2 ; \s__fp \__fp_chk:w #3 #4
13549   { \__fp_exp_after_o:w \s__fp \__fp_chk:w #3 #1 }
```

(*End 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.

```
13550 \cs_new:Npn \__fp_add_zeros_o:Nww #1 \s__fp \__fp_chk:w 0 #2
13551   {
13552     \if_int_compare:w #2 #1 = 20 \exp_stop_f:
13553       \exp_after:wN \__fp_add_return_ii_o:Nww
13554     \else:
13555       \__fp_case_return_i_o:ww
13556     \fi:
13557     #1
13558     \s__fp \__fp_chk:w 0 #2
13559   }
```

(End 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 ⟨*sign₂*⟩ (#1) and the ⟨*sign₂*⟩ (#4) are identical.

```
13560 \cs_new:Npn \__fp_add_inf_o:Nww
13561     #1 \s__fp \__fp_chk:w 2 #2 #3; \s__fp \__fp_chk:w 2 #4
13562   {
13563     \if_meaning:w #1 #2
13564       \__fp_case_return_i_o:ww
13565     \else:
13566       \__fp_case_use:nw
13567         {
13568           \if_meaning:w #1 #4
13569             \exp_after:wN \__fp_invalid_operation_o:Nww
13570             \exp_after:wN +
13571           \else:
13572             \exp_after:wN \__fp_invalid_operation_o:Nww
13573             \exp_after:wN -
13574           \fi:
13575         }
13576     \fi:
13577     \s__fp \__fp_chk:w 2 #2 #3;
13578     \s__fp \__fp_chk:w 2 #4
13579   }
```

(End definition for `\__fp_add_inf_o:Nww`.*)*

`\__fp_add_normal_o:Nww`   `\__fp_add_normal_o:Nww` ⟨*sign₂*⟩ `\s__fp \__fp_chk:w 1` ⟨*sign₁*⟩ ⟨*exp₁*⟩ ⟨*body₁*⟩ ; `\s__fp \__fp_chk:w 1` ⟨*initial sign₂*⟩ ⟨*exp₂*⟩ ⟨*body₂*⟩ ;

We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.

```
13580 \cs_new:Npn \__fp_add_normal_o:Nww #1 \s__fp \__fp_chk:w 1 #2
13581   {
13582     \if_meaning:w #1#2
13583       \exp_after:wN \__fp_add_npos_o:NnwNnw
13584     \else:
13585       \exp_after:wN \__fp_sub_npos_o:NnwNnw
13586     \fi:
13587     #2
13588   }
```

(End definition for `\__fp_add_normal_o:Nww`.*)*

27.2.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

`\__fp_add_npos_o:NnwNnw`   `\__fp_add_npos_o:NnwNnw` ⟨*sign₁*⟩ ⟨*exp₁*⟩ ⟨*body₁*⟩ ; `\s__fp \__fp_chk:w 1` ⟨*initial sign₂*⟩ ⟨*exp₂*⟩ ⟨*body₂*⟩ ;

Since we are doing an addition, the final sign is ⟨*sign₁*⟩. Start an `\__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.

```
13589 \cs_new:Npn \__fp_add_npos_o:NnwNnw #1#2#3 ; \s__fp \__fp_chk:w 1 #4 #5
13590   {
13591     \exp_after:wN \__fp_sanitize:Nw
13592     \exp_after:wN #1
13593     \__int_value:w \__int_eval:w
13594       \if_int_compare:w #2 > #5 \exp_stop_f:
13595         #2
13596         \exp_after:wN \__fp_add_big_i_o:wNww \__int_value:w -
13597       \else:
13598         #5
13599         \exp_after:wN \__fp_add_big_ii_o:wNww \__int_value:w
13600       \fi:
13601       \__int_eval:w #5 - #2 ; #1 #3;
13602   }
```

(*End definition for* `\__fp_add_npos_o:NnwNnw`.)

`\__fp_add_big_i_o:wNww`   `\__fp_add_big_i_o:wNww` ⟨shift⟩ ; ⟨final sign⟩ ⟨body₁⟩ ; ⟨body₂⟩ ;
`\__fp_add_big_ii_o:wNww`
    Shift the significand of the small number, then add with `\__fp_add_significand_-o:NnnwnnnnN`.

```
13603 \cs_new:Npn \__fp_add_big_i_o:wNww #1; #2 #3; #4;
13604   {
13605     \__fp_decimate:nNnnnn {#1}
13606       \__fp_add_significand_o:NnnwnnnnN
13607       #4
13608     #3
13609     #2
13610   }
13611 \cs_new:Npn \__fp_add_big_ii_o:wNww #1; #2 #3; #4;
13612   {
13613     \__fp_decimate:nNnnnn {#1}
13614       \__fp_add_significand_o:NnnwnnnnN
13615       #3
13616     #4
13617     #2
13618   }
```

(*End definition for* `\__fp_add_big_i_o:wNww` *and* `\__fp_add_big_ii_o:wNww`.)

`\__fp_add_significand_o:NnnwnnnnN`   `\__fp_add_significand_o:NnnwnnnnN` ⟨rounding digit⟩ {⟨Y'_1⟩} {⟨Y'_2⟩}
`\__fp_add_significand_pack:NNNNNNN`   ⟨extra-digits⟩ ; {⟨X_1⟩} {⟨X_2⟩} {⟨X_3⟩} {⟨X_4⟩} ⟨final sign⟩
`\__fp_add_significand_test_o:N`
    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.

```
13619 \cs_new:Npn \__fp_add_significand_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
13620   {
13621     \exp_after:wN \__fp_add_significand_test_o:N
13622     \__int_value:w \__int_eval:w 1#5#6 + #2
```

```
13623        \exp_after:wN \__fp_add_significand_pack:NNNNNNN
13624        \__int_value:w \__int_eval:w 1#7#8 + #3 ; #1
13625    }
13626 \cs_new:Npn \__fp_add_significand_pack:NNNNNNN #1 #2#3#4#5#6#7
13627    {
13628      \if_meaning:w 2 #1
13629        + \c_one
13630      \fi:
13631      ; #2 #3 #4 #5 #6 #7 ;
13632    }
13633 \cs_new:Npn \__fp_add_significand_test_o:N #1
13634    {
13635      \if_meaning:w 2 #1
13636        \exp_after:wN \__fp_add_significand_carry_o:wwwNN
13637      \else:
13638        \exp_after:wN \__fp_add_significand_no_carry_o:wwwNN
13639      \fi:
13640    }
```

(*End 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.

```
13641 \cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
13642      #1; #2; #3#4 ; #5#6
13643    {
13644      \exp_after:wN \__fp_basics_pack_high:NNNNNw
13645      \__int_value:w \__int_eval:w 1 #1
13646        \exp_after:wN \__fp_basics_pack_low:NNNNNw
13647        \__int_value:w \__int_eval:w 1 #2 #3#4
13648          + \__fp_round:NNN #6 #4 #5
13649          \exp_after:wN ;
13650    }
```

(*End 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.

```
13651 \cs_new:Npn \__fp_add_significand_carry_o:wwwNN
13652      #1; #2; #3#4; #5#6
13653    {
13654      + \c_one
13655      \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw
13656      \__int_value:w \__int_eval:w 1 1 #1
13657        \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
13658        \__int_value:w \__int_eval:w 1 #2#3 +
13659          \exp_after:wN \__fp_round:NNN
13660          \exp_after:wN #6
13661          \exp_after:wN #3
```

```
13662                \__int_value:w \__fp_round_digit:Nw #4 #5 ;
13663                \exp_after:wN ;
13664        }
```

(*End definition for* `\__fp_add_significand_carry_o:wwwNN`.)

27.2.3 Absolute subtraction

$\__fp_sub_npos_o:NnwNnw$ $\langle sign_1 \rangle$ $\langle exp_1 \rangle$ $\langle body_1 \rangle$; \s__fp __fp_chk:w 1
$\langle initial\ sign_2 \rangle$ $\langle exp_2 \rangle$ $\langle body_2 \rangle$;

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 $\langle sign_1 \rangle$.

```
13665 \cs_new:Npn \__fp_sub_npos_o:NnwNnw #1#2#3; \s__fp \__fp_chk:w 1 #4#5#6;
13666    {
13667      \if_case:w \__fp_compare_npos:nwnw {#2} #3; {#5} #6; \exp_stop_f:
13668        \exp_after:wN \__fp_sub_eq_o:Nnwnw
13669      \or:
13670        \exp_after:wN \__fp_sub_npos_i_o:Nnwnw
13671      \else:
13672        \exp_after:wN \__fp_sub_npos_ii_o:Nnwnw
13673      \fi:
13674      #1 {#2} #3; {#5} #6;
13675    }
13676 \cs_new:Npn \__fp_sub_eq_o:Nnwnw #1#2; #3; { \exp_after:wN \c_zero_fp }
13677 \cs_new:Npn \__fp_sub_npos_ii_o:Nnwnw #1 #2; #3;
13678    {
13679      \exp_after:wN \__fp_sub_npos_i_o:Nnwnw
13680        \__int_value:w \__int_eval:w \c_two - #1 \__int_eval_end:
13681        #3; #2;
13682    }
```

(*End definition for* `\__fp_sub_npos_o:NnwNnw`, `\__fp_sub_eq_o:Nnwnw`, *and* `\__fp_sub_npos_ii_o:Nnwnw`.)

After the computation is done, __fp_sanitize:Nw checks for overflow/underflow. It expects the $\langle final\ sign \rangle$ and the $\langle exponent \rangle$ (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.

```
13683 \cs_new:Npn \__fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
13684    {
13685      \exp_after:wN \__fp_sanitize:Nw
13686      \exp_after:wN #1
13687      \__int_value:w \__int_eval:w
13688        #2
13689        \if_int_compare:w #2 = #4 \exp_stop_f:
13690          \exp_after:wN \__fp_sub_back_near_o:nnnnnnnnN
13691        \else:
13692          \exp_after:wN \__fp_decimate:nNnnnn \exp_after:wN
13693            { \__int_value:w \__int_eval:w #2 - #4 - \c_one \exp_after:wN }
```

```
13694              \exp_after:wN \__fp_sub_back_far_o:NnnwnnnnN
13695        \fi:
13696          #5
13697        #3
13698        #1
13699    }
```

(*End definition for* `\__fp_sub_npos_i_o:Nnwnw.`)

$\__fp\_sub\_back\_near\_o:nnnnnnnnN$ {$\langle Y_1 \rangle$} {$\langle Y_2 \rangle$} {$\langle Y_3 \rangle$} {$\langle Y_4 \rangle$} {$\langle X_1 \rangle$}
{$\langle X_2 \rangle$} {$\langle X_3 \rangle$} {$\langle X_4 \rangle$} $\langle$*final sign*$\rangle$

In this case, the subtraction is exact, so we discard the $\langle$*final sign*$\rangle$ #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.

```
13700 \cs_new:Npn \__fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
13701   {
13702     \exp_after:wN \__fp_sub_back_near_after:wNNNNw
13703     \__int_value:w \__int_eval:w 10#5#6 - #1#2 - \c_eleven
13704       \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
13705         \__int_value:w \__int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
13706   }
13707 \cs_new:Npn \__fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
13708   { + #1#2 ; {#3#4#5#6} {#7} ; }
13709 \cs_new:Npn \__fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 ;
13710   {
13711     \if_meaning:w 0 #1
13712       \exp_after:wN \__fp_sub_back_shift:wnnnn
13713     \fi:
13714     ; {#1#2#3#4} {#5}
13715   }
```

(*End definition for* `\__fp_sub_back_near_o:nnnnnnnnN, \__fp_sub_back_near_pack:NNNNNNw, and \__-`
`fp_sub_back_near_after:wNNNNw.`)

$\__fp\_sub\_back\_shift:wnnnn$; {$\langle Z_1 \rangle$} {$\langle Z_2 \rangle$} {$\langle Z_3 \rangle$} {$\langle Z_4 \rangle$} ;

This function is called with $\langle Z_1 \rangle \leq 999$. Act with \number to trim leading zeros from
$\langle Z_1 \rangle$ $\langle Z_2 \rangle$ (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.

```
13716 \cs_new:Npn \__fp_sub_back_shift:wnnnn ; #1#2
13717   {
13718     \exp_after:wN \__fp_sub_back_shift_ii:ww
13719       \__int_value:w #1 #2 0 ;
13720   }
13721 \cs_new:Npn \__fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
13722   {
13723     \if_meaning:w @ #1 @
13724       - \c_seven
```

```
13725        - \exp_after:wN \use_i:nnn
13726          \exp_after:wN \__fp_sub_back_shift_iii:NNNNNNNNNw
13727          \__int_value:w #2#3 0 ~ 123456789;
13728      \else:
13729        - \__fp_sub_back_shift_iii:NNNNNNNNNw #1 123456789;
13730      \fi:
13731      \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
13732      \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
13733      \exp_after:wN \__fp_sub_back_shift_iv:nnnnw
13734      \exp_after:wN ;
13735      \__int_value:w
13736      #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
13737    }
13738 \cs_new:Npn \__fp_sub_back_shift_iii:NNNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
13739 \cs_new:Npn \__fp_sub_back_shift_iv:nnnnw #1 ; #2 ; { ; #1 ; }
```

(*End 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 $10^{⟨expo_x⟩}$, call the `very_far` auxiliary. If the result is less than $10^{⟨expo_x⟩}$, call the `not_far` auxiliary. If it is too close a call to know yet, namely if $1⟨Y'_1⟩⟨Y'_2⟩ = ⟨X_1⟩⟨X_2⟩⟨X_3⟩⟨X_4⟩0$, 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).

```
13740 \cs_new:Npn \__fp_sub_back_far_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
13741    {
13742      \if_case:w
13743        \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
13744          \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
13745            \c_zero
13746          \else:
13747            \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: \c_one
13748          \fi:
13749        \else:
13750          \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: \c_one
13751        \fi:
13752            \exp_after:wN \__fp_sub_back_quite_far_o:wwNN
13753      \or:   \exp_after:wN \__fp_sub_back_very_far_o:wwwwNN
13754      \else: \exp_after:wN \__fp_sub_back_not_far_o:wwwwNN
13755      \fi:
13756      #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1
13757    }
```

(*End definition for* `\__fp_sub_back_far_o:NnnwnnnnN.`)

`\__fp_sub_back_quite_far_o:wwNN`   The easiest case is when $x - y$ is extremely close to a power of 10, namely the first digit
`\__fp_sub_back_quite_far_ii:NN`   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 will 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).

```
13758 \cs_new:Npn \__fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
13759    {
```

```
13760        \exp_after:wN \__fp_sub_back_quite_far_ii:NN
13761        \exp_after:wN #3
13762        \exp_after:wN #4
13763      }
13764 \cs_new:Npn \__fp_sub_back_quite_far_ii:NN #1#2
13765    {
13766      \if_case:w \__fp_round_neg:NNN #2 0 #1
13767        \exp_after:wN \use_i:nn
13768      \else:
13769        \exp_after:wN \use_ii:nn
13770      \fi:
13771        { ; {1000} {0000} {0000} {0000} ; }
13772        { - \c_one ; {9999} {9999} {9999} {9999} ; }
13773    }
```

(*End 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 `- \c_one`). 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`.

```
13774 \cs_new:Npn \__fp_sub_back_not_far_o:wwwwNN #1 ~ #2; #3 ~ #4; #5#6
13775    {
13776      - \c_one
13777      \exp_after:wN \__fp_sub_back_near_after:wNNNNw
13778      \__int_value:w \__int_eval:w 1#30 - #1 - \c_eleven
13779        \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
13780        \__int_value:w \__int_eval:w 11 0000 0000 + #40 - #2
13781          - \exp_after:wN \__fp_round_neg:NNN
13782            \exp_after:wN #6
13783            \use_none:nnnnnnn #2 #5
13784          \exp_after:wN ;
13785    }
```

(*End 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.

```
13786 \cs_new:Npn \__fp_sub_back_very_far_o:wwwwNN #1#2#3#4#5#6#7
13787    {
13788      \__fp_pack_eight:wNNNNNNNN
13789      \__fp_sub_back_very_far_ii_o:nnNwwNN
13790        { 0 #1#2#3 #4#5#6#7 }
```

```
13791        ;
13792    }
13793  \cs_new:Npn \__fp_sub_back_very_far_ii_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
13794    {
13795      \exp_after:wN \__fp_basics_pack_high:NNNNNw
13796      \__int_value:w \__int_eval:w 1#4 - #1 - \c_one
13797        \exp_after:wN \__fp_basics_pack_low:NNNNNw
13798        \__int_value:w \__int_eval:w 2#5 - #2
13799          - \exp_after:wN \__fp_round_neg:NNN
13800            \exp_after:wN #7
13801            \__int_value:w
13802              \if_int_odd:w \__int_eval:w #5 - #2 \__int_eval_end:
13803                1 \else: 2 \fi:
13804            \__int_value:w \__fp_round_digit:Nw #3 #6 ;
13805      \exp_after:wN ;
13806    }
```

(*End definition for* __fp_sub_back_very_far_o:wwwwNN *and* __fp_sub_back_very_far_ii_o:nnNwwNN.)

27.3 Multiplication

27.3.1 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.

```
13807  \cs_new:cpn { __fp_*_o:ww }
13808    {
13809      \__fp_mul_cases_o:NnNnww
13810        *
13811        { - \c_two + }
13812        \__fp_mul_npos_o:Nww
13813        { }
13814    }
```

(*End 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
$0\times0 = 0\times1 = 1\times0 = 0$ to case 4 or 5 depending on the combined sign, the products $0\times\infty$
and $\infty\times0$ to case 6 or 7 (invalid operation), and the products $1\times\infty = \infty\times1 = \infty\times\infty = \infty$
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.

```
13815  \cs_new:Npn \__fp_mul_cases_o:NnNnww
13816      #1#2#3#4 \s__fp \__fp_chk:w #5#6#7; \s__fp \__fp_chk:w #8#9
13817    {
```

629

```
13818        \if_case:w \__int_eval:w
13819                  \if_int_compare:w #5 #8 = \c_eleven
13820                    \c_one
13821                  \else:
13822                    \if_meaning:w 3 #8
13823                      \c_three
13824                    \else:
13825                      \if_meaning:w 3 #5
13826                        \c_two
13827                      \else:
13828                        \if_int_compare:w #5 #8 = \c_ten
13829                          \c_nine #2 - \c_two
13830                        \else:
13831                          (#5 #2 #8) / \c_two * \c_two + \c_seven
13832                        \fi:
13833                      \fi:
13834                    \fi:
13835                  \fi:
13836                  \if_meaning:w #6 #9 - \c_one \fi:
13837                \__int_eval_end:
13838          \__fp_case_use:nw { #3 0 }
13839      \or: \__fp_case_use:nw { #3 2 }
13840      \or: \__fp_case_return_i_o:ww
13841      \or: \__fp_case_return_ii_o:ww
13842      \or: \__fp_case_return_o:Nww \c_zero_fp
13843      \or: \__fp_case_return_o:Nww \c_minus_zero_fp
13844      \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
13845      \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
13846      \or: \__fp_case_return_o:Nww \c_inf_fp
13847      \or: \__fp_case_return_o:Nww \c_minus_inf_fp
13848      #4
13849      \fi:
13850      \s__fp \__fp_chk:w #5 #6 #7;
13851      \s__fp \__fp_chk:w #8 #9
13852    }
```

(*End definition for* __fp_mul_cases_o:nNnnww.)

27.3.2 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 ⟨sign₁⟩ {⟨exp₁⟩}
⟨body₁⟩ ; \s__fp __fp_chk:w 1 ⟨sign₂⟩ {⟨exp₂⟩} ⟨body₂⟩ ;

After the computation, __fp_sanitize:Nw checks for overflow or underflow. As we did for addition, __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.

```
13853 \cs_new:Npn \__fp_mul_npos_o:Nww
13854    #1 \s__fp \__fp_chk:w #2 #3 #4 #5 ; \s__fp \__fp_chk:w #6 #7 #8 #9 ;
13855    {
13856      \exp_after:wN \__fp_sanitize:Nw
13857      \exp_after:wN #1
```

```
13858        \__int_value:w \__int_eval:w
13859          #4 + #8
13860          \__fp_mul_significand_o:nnnnNnnnn #5 #1 #9
13861      }
```

(*End definition for* `\__fp_mul_npos_o:Nww`.)

`\__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 `\__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 `\__int_eval:w`.

```
13862 \cs_new:Npn \__fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
13863   {
13864     \exp_after:wN \__fp_mul_significand_test_f:NNN
13865     \exp_after:wN #5
13866     \__int_value:w \__int_eval:w 99990000 + #1*#6 +
13867       \exp_after:wN \__fp_mul_significand_keep:NNNNNw
13868       \__int_value:w \__int_eval:w 99990000 + #1*#7 + #2*#6 +
13869         \exp_after:wN \__fp_mul_significand_keep:NNNNNw
13870         \__int_value:w \__int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
13871           \exp_after:wN \__fp_mul_significand_drop:NNNNNw
13872           \__int_value:w \__int_eval:w 99990000 + #1*#9 + #2*#8 + #3*#7 + #4*#6 +
13873             \exp_after:wN \__fp_mul_significand_drop:NNNNNw
13874             \__int_value:w \__int_eval:w 99990000 + #2*#9 + #3*#8 + #4*#7 +
13875               \exp_after:wN \__fp_mul_significand_drop:NNNNNw
13876               \__int_value:w \__int_eval:w 99990000 + #3*#9 + #4*#8 +
13877                 \exp_after:wN \__fp_mul_significand_drop:NNNNNw
13878                 \__int_value:w \__int_eval:w 100000000 + #4*#9 ;
13879     ; \exp_after:wN ;
13880   }
13881 \cs_new:Npn \__fp_mul_significand_drop:NNNNNw #1#2#3#4#5 #6;
13882   { #1#2#3#4#5 ; + #6 }
13883 \cs_new:Npn \__fp_mul_significand_keep:NNNNNw #1#2#3#4#5 #6;
13884   { #1#2#3#4#5 ; #6 ; }
```

(*End definition for* `\__fp_mul_significand_o:nnnnNnnnn`, `\__fp_mul_significand_drop:NNNNNw`, *and* `\__fp_mul_significand_keep:NNNNNw`.)

`\__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.

```
13885 \cs_new:Npn \__fp_mul_significand_test_f:NNN #1 #2 #3
```

```
13886    {
13887      \if_meaning:w 0 #3
13888        \exp_after:wN \__fp_mul_significand_small_f:NNwwwN
13889      \else:
13890        \exp_after:wN \__fp_mul_significand_large_f:NwwNNNN
13891      \fi:
13892      #1 #3
13893    }
```

(*End 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`.

```
13894  \cs_new:Npn \__fp_mul_significand_large_f:NwwNNNN #1 #2; #3; #4#5#6#7; +
13895    {
13896      \exp_after:wN \__fp_basics_pack_high:NNNNNw
13897      \__int_value:w \__int_eval:w 1#2
13898        \exp_after:wN \__fp_basics_pack_low:NNNNNw
13899        \__int_value:w \__int_eval:w 1#3#4#5#6#7
13900          + \exp_after:wN \__fp_round:NNN
13901            \exp_after:wN #1
13902            \exp_after:wN #7
13903            \__int_value:w \__fp_round_digit:Nw
13904    }
```

(*End definition for* `\__fp_mul_significand_large_f:NwwNNNN.`)

`\__fp_mul_significand_small_f:NNwwwN` In this branch, ⟨*digit 1*⟩ is zero. Our result will thus be ⟨*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.

```
13905  \cs_new:Npn \__fp_mul_significand_small_f:NNwwwN #1 #2#3; #4#5; #6; + #7
13906    {
13907      - \c_one
13908      \exp_after:wN \__fp_basics_pack_high:NNNNNw
13909      \__int_value:w \__int_eval:w 1#3#4
13910        \exp_after:wN \__fp_basics_pack_low:NNNNNw
13911        \__int_value:w \__int_eval:w 1#5#6#7
13912          + \exp_after:wN \__fp_round:NNN
13913            \exp_after:wN #1
13914            \exp_after:wN #7
13915            \__int_value:w \__fp_round_digit:Nw
13916    }
```

(*End definition for* `\__fp_mul_significand_small_f:NNwwwN.`)

27.4 Division

27.4.1 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 - `\c_two` + 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.

```
13917 \cs_new:cpn { __fp_/_o:ww }
13918   {
13919     \__fp_mul_cases_o:NnNnww
13920       /
13921       { - }
13922       \__fp_div_npos_o:Nww
13923       {
13924         \or:
13925           \__fp_case_use:nw
13926             { \__fp_division_by_zero_o:NNww \c_inf_fp / }
13927         \or:
13928           \__fp_case_use:nw
13929             { \__fp_division_by_zero_o:NNww \c_minus_inf_fp / }
13930       }
13931   }
```

(*End 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.

```
13932 \cs_new:Npn \__fp_div_npos_o:Nww
13933   #1 \s__fp \__fp_chk:w 1 #2 #3 #4 ; \s__fp \__fp_chk:w 1 #5 #6 #7#8#9;
13934   {
13935     \exp_after:wN \__fp_sanitize:Nw
13936     \exp_after:wN #1
13937     \__int_value:w \__int_eval:w
13938       #3 - #6
13939       \exp_after:wN \__fp_div_significand_i_o:wnnw
13940         \__int_value:w \__int_eval:w #7 \use_i:nnnn #8 + \c_one ;
13941       #4
13942       {#7}{#8}#9 ;
13943       #1
13944   }
```

(*End definition for* `\__fp_div_npos_o:Nww`.)

27.4.2 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 $Q_A \simeq 10^4 A/Z$.

- Replace A by $B = 10^4 A - Q_A Z$.

- Find an integer $Q_B \simeq 10^4 B/Z$.

- Replace B by $C = 10^4 B - Q_B Z$.

- Find an integer $Q_C \simeq 10^4 C/Z$.

- Replace C by $D = 10^4 C - Q_C Z$.

- Find an integer $Q_D \simeq 10^4 D/Z$.

- Consider $E = 10^4 D - Q_D Z$, and ensure correct rounding.

The result is then $Q = 10^{-4} Q_A + 10^{-8} Q_B + 10^{-12} Q_C + 10^{-16} Q_D +$ 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 will be 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

$$\texttt{\textbackslash int\_eval:n} \left\{ \frac{A_1 A_2}{Z_1 + 1} - 1 \right\} \leq 10^4 \frac{A}{Z}$$

Subtracting 1 at the end takes care of the fact that ε-TeX's $\texttt{\textbackslash\_\_int\_eval:w}$ 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

$$Q_A = \texttt{\textbackslash int\_eval:n} \left\{ \frac{10 \cdot A_1 A_2}{\lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1} - 1 \right\}.$$

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

$$y = \lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1,$$

so that, taking into account the fact that ε-TeX rounds ties away from zero,

$$Q_A = \left\lfloor \frac{A_1 A_2 0}{y} - \frac{1}{2} \right\rfloor$$
$$> \frac{A_1 A_2 0}{y} - \frac{3}{2}.$$

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:

$$10^5 B = A_1 A_2 0 + 10 \cdot 0.A_3 A_4 - 10 \cdot Z_1.Z_2 Z_3 Z_4 \cdot Q_A$$
$$< A_1 A_2 0 \cdot \left(1 - 10 \cdot \frac{Z_1.Z_2 Z_3 Z_4}{y} \right) + \frac{3}{2} \cdot 10 \cdot Z_1.Z_2 Z_3 Z_4 + 10$$
$$\leq \frac{A_1 A_2 0 \cdot (y - 10 \cdot Z_1.Z_2 Z_3 Z_4)}{y} + \frac{3}{2} y + 10$$
$$\leq \frac{A_1 A_2 0 \cdot 1}{y} + \frac{3}{2} y + 10 \leq \frac{10^9 A}{y} + 1.6 \cdot y.$$

At the last step, we hide 10 into the second term for later convenience. The same reasoning yields

$$10^5 B < 10^9 A / y + 1.6 y,$$
$$10^5 C < 10^9 B / y + 1.6 y,$$
$$10^5 D < 10^9 C / y + 1.6 y,$$
$$10^5 E < 10^9 D / y + 1.6 y.$$

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

$$10^5 B < 10^9 / y + 1.6 y,$$
$$10^5 C < 10^{13} / y^2 + 1.6(y + 10^4),$$
$$10^5 D < 10^{17} / y^3 + 1.6(y + 10^4 + 10^8 / y),$$
$$10^5 E < 10^{21} / y^4 + 1.6(y + 10^4 + 10^8 / y + 10^{12} / y^2).$$

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,

$$10^5 B < \max(1.16 \cdot 10^5, 1.7 \cdot 10^5),$$
$$10^5 C < \max(1.32 \cdot 10^5, 1.77 \cdot 10^5),$$
$$10^5 D < \max(1.48 \cdot 10^5, 1.777 \cdot 10^5),$$
$$10^5 E < \max(1.64 \cdot 10^5, 1.7777 \cdot 10^5).$$

All of those bounds are less than $2.147 \cdot 10^5$, and we are thus within TEX's bounds in all cases!

We will 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 will be 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

$$\frac{2E}{Z} = 2\frac{10^5 E}{10^5 Z} \le 2\frac{10^5 E}{10^4} < 36,$$

this integer part is between 0 and 35 inclusive. We let ε-TEX round

$$P = \texttt{\char`\\int\_eval:n} \left\{ \frac{2 \cdot E_1 E_2}{Z_1 Z_2} \right\},$$

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 $E/Z \in \left((P-1)/2, P/2\right)$. If it is zero, then $E/Z = P/2$. If it is positive, then $E/Z \in \left(P/2, (P-1)/2\right)$. In each case, we know how to round to an integer, depending on the parity of P, and the rounding mode.

27.4.3 Implementing the significand division

`\_fp_div_significand_i_o:wnnw`

$\texttt{\char`\\__fp\_div\_significand\_i\_o:wnnw}$ $\langle y \rangle$; $\{\langle A_1 \rangle\}$ $\{\langle A_2 \rangle\}$ $\{\langle A_3 \rangle\}$ $\{\langle A_4 \rangle\}$ $\{\langle Z_1 \rangle\}$ $\{\langle Z_2 \rangle\}$ $\{\langle Z_3 \rangle\}$ $\{\langle Z_4 \rangle\}$; $\langle sign \rangle$

Compute $10^6 + Q_A$ (a 7 digit number thanks to the shift), unbrace $\langle A_1 \rangle$ and $\langle A_2 \rangle$, and prepare the $\langle continuation \rangle$ arguments for 4 consecutive calls to $\texttt{\char`\\__fp\_div\_significand\_calc:wwnnnnnnn}$. Each of these calls will need $\langle y \rangle$ (#1), and it turns out that we need post-expansion there, hence the $\texttt{\char`\\__int\_value:w}$. Here, #4 is six brace groups, which give the six first n-type arguments of the calc function.

```
13945 \cs_new:Npn \__fp_div_significand_i_o:wnnw #1 ; #2#3 #4 ;
13946   {
13947     \exp_after:wN \__fp_div_significand_test_o:w
13948     \__int_value:w \__int_eval:w
13949       \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
13950       \__int_value:w \__int_eval:w 999999 + #2 #3 0 / #1 ;
13951         #2 #3 ;
13952         #4
```

```
13953          { \exp_after:wN \__fp_div_significand_ii:wwn \__int_value:w #1 }
13954          { \exp_after:wN \__fp_div_significand_ii:wwn \__int_value:w #1 }
13955          { \exp_after:wN \__fp_div_significand_ii:wwn \__int_value:w #1 }
13956          { \exp_after:wN \__fp_div_significand_iii:wwnnnnn \__int_value:w #1 }
13957      }
```

(*End 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`

$\__fp_div_significand_calc:wwnnnnnnn$ $\langle 10^6 + Q_A \rangle$; $\langle A_1 \rangle \langle A_2 \rangle$; $\{\langle A_3 \rangle\}$ $\{\langle A_4 \rangle\}$ $\{\langle Z_1 \rangle\}$ $\{\langle Z_2 \rangle\}$ $\{\langle Z_3 \rangle\}$ $\{\langle Z_4 \rangle\}$ $\{\langle continuation \rangle\}$
expands to

$\langle 10^6 + Q_A \rangle$ $\langle continuation \rangle$; $\langle B_1 \rangle \langle B_2 \rangle$; $\{\langle B_3 \rangle\}$ $\{\langle B_4 \rangle\}$ $\{\langle Z_1 \rangle\}$ $\{\langle Z_2 \rangle\}$ $\{\langle Z_3 \rangle\}$ $\{\langle Z_4 \rangle\}$

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.

```
13958 \cs_new:Npn \__fp_div_significand_calc:wwnnnnnnn 1#1
13959   {
13960     \if_meaning:w 1 #1
13961       \exp_after:wN \__fp_div_significand_calc_i:wwnnnnnnn
13962     \else:
13963       \exp_after:wN \__fp_div_significand_calc_ii:wwnnnnnnn
13964     \fi:
13965   }
13966 \cs_new:Npn \__fp_div_significand_calc_i:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
13967   {
13968     1 1 #1
```

```
13969        #9 \exp_after:wN ;
13970        \__int_value:w \__int_eval:w \c__fp_Bigg_leading_shift_int
13971          + #2 - #1 * #5 - #5#60
13972          \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13973          \__int_value:w \__int_eval:w \c__fp_Bigg_middle_shift_int
13974            + #3 - #1 * #6 - #70
13975          \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13976          \__int_value:w \__int_eval:w \c__fp_Bigg_middle_shift_int
13977            + #4 - #1 * #7 - #80
13978          \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13979          \__int_value:w \__int_eval:w \c__fp_Bigg_trailing_shift_int
13980            - #1 * #8 ;
13981        {#5}{#6}{#7}{#8}
13982      }
13983  \cs_new:Npn \__fp_div_significand_calc_ii:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
13984    {
13985      1 0 #1
13986      #9 \exp_after:wN ;
13987      \__int_value:w \__int_eval:w \c__fp_Bigg_leading_shift_int
13988        + #2 - #1 * #5
13989        \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13990        \__int_value:w \__int_eval:w \c__fp_Bigg_middle_shift_int
13991          + #3 - #1 * #6
13992        \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13993        \__int_value:w \__int_eval:w \c__fp_Bigg_middle_shift_int
13994          + #4 - #1 * #7
13995        \exp_after:wN \__fp_pack_Bigg:NNNNNNw
13996        \__int_value:w \__int_eval:w \c__fp_Bigg_trailing_shift_int
13997          - #1 * #8 ;
13998      {#5}{#6}{#7}{#8}
13999    }
```

(*End 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 $\langle y \rangle$; $\langle B_1 \rangle$; {$\langle B_2 \rangle$} {$\langle B_3 \rangle$} {$\langle B_4 \rangle$} {$\langle Z_1 \rangle$}
{$\langle Z_2 \rangle$} {$\langle Z_3 \rangle$} {$\langle Z_4 \rangle$} $\langle continuations \rangle$ $\langle sign \rangle$

Compute Q_B by evaluating $\langle B_1 \rangle\langle B_2 \rangle 0/y - 1$. The result will be output to the left, in an __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.

```
14000  \cs_new:Npn \__fp_div_significand_ii:wwn #1; #2;#3
14001    {
14002      \exp_after:wN \__fp_div_significand_pack:NNN
14003      \__int_value:w \__int_eval:w
14004        \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
14005        \__int_value:w \__int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
14006    }
```

(*End definition for* __fp_div_significand_ii:wwn.)

__fp_div_significand_iii:wwnnnnn   __fp_div_significand_iii:wwnnnnn $\langle y \rangle$; $\langle E_1 \rangle$; {$\langle E_2 \rangle$} {$\langle E_3 \rangle$} {$\langle E_4 \rangle$}
{$\langle Z_1 \rangle$} {$\langle Z_2 \rangle$} {$\langle Z_3 \rangle$} {$\langle Z_4 \rangle$} $\langle sign \rangle$

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 will 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.

```
14007 \cs_new:Npn \__fp_div_significand_iii:wwnnnnn #1; #2;#3#4#5 #6#7
14008   {
14009     0
14010     \exp_after:wN \__fp_div_significand_iv:wwnnnnnnn
14011     \__int_value:w \__int_eval:w (\c_two * #2 #3) / #6 #7 ; % <- P
14012       #2 ; {#3} {#4} {#5}
14013       {#6} {#7}
14014   }
```

(*End definition for* `\__fp_div_significand_iii:wwnnnnn`*.*)

<div style="float:left">

`\__fp_div_significand_iv:wwnnnnnnn`
`\__fp_div_significand_v:NNw`
`\__fp_div_significand_vi:Nw`

</div>

$\__fp_div_significand_iv:wwnnnnnnn$ $\langle P\rangle$; $\langle E_1\rangle$; $\{\langle E_2\rangle\}$ $\{\langle E_3\rangle\}$ $\{\langle E_4\rangle\}$ $\{\langle Z_1\rangle\}$ $\{\langle Z_2\rangle\}$ $\{\langle Z_3\rangle\}$ $\{\langle Z_4\rangle\}$ $\langle sign\rangle$

This adds to the current expression $(10^7 + 10 \cdot Q_D)$ a contribution of $5 \cdot P + \text{sign}(T)$ with $T = 2E - PZ$. This amounts to adding $P/2$ to Q_D, with an extra $\langle rounding\rangle$ digit. This $\langle rounding\rangle$ 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: $T = 0$.

```
14015 \cs_new:Npn \__fp_div_significand_iv:wwnnnnnnn #1; #2;#3#4#5 #6#7#8#9
14016   {
14017     + \c_five * #1
14018     \exp_after:wN \__fp_div_significand_vi:Nw
14019     \__int_value:w \__int_eval:w -20 + 2*#2#3 - #1*#6#7 +
14020       \exp_after:wN \__fp_div_significand_v:NN
14021       \__int_value:w \__int_eval:w 199980 + 2*#4 - #1*#8 +
14022         \exp_after:wN \__fp_div_significand_v:NN
14023         \__int_value:w \__int_eval:w 200000 + 2*#5 - #1*#9 ;
14024   }
14025 \cs_new:Npn \__fp_div_significand_v:NN #1#2 { #1#2 \__int_eval_end: + }
14026 \cs_new:Npn \__fp_div_significand_vi:Nw #1#2;
```

```
14027        {
14028          \if_meaning:w 0 #1
14029            \if_int_compare:w \__int_eval:w #2 > \c_zero + \c_one \fi:
14030          \else:
14031            \if_meaning:w - #1 - \else: + \fi: \c_one
14032          \fi:
14033          ;
14034        }
```

(*End 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` $10^7 + 10 \cdot Q_D + 5 \cdot P + \varepsilon$; $\langle sign \rangle$

Here, $\varepsilon = \text{sign}(T)$ 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).

```
14035 \cs_new:Npn \__fp_div_significand_pack:NNN 1 #1 #2 { + #1 #2 ; }
```

(*End definition for* `\__fp_div_significand_pack:NNN`.)

`\__fp_div_significand_test_o:w`   `\__fp_div_significand_test_o:w` 1 0 $\langle 5d \rangle$; $\langle 4d \rangle$; $\langle 4d \rangle$; $\langle 5d \rangle$; $\langle sign \rangle$
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.

```
14036 \cs_new:Npn \__fp_div_significand_test_o:w 10 #1
14037   {
14038     \if_meaning:w 0 #1
14039       \exp_after:wN \__fp_div_significand_small_o:wwwNNNNwN
14040     \else:
14041       \exp_after:wN \__fp_div_significand_large_o:wwwNNNNwN
14042     \fi:
14043     #1
14044   }
```

(*End definition for* `\__fp_div_significand_test_o:w`.)

`\__fp_div_significand_small_o:wwwNNNNwN`   `\__fp_div_significand_small_o:wwwNNNNwN` 0 $\langle 4d \rangle$; $\langle 4d \rangle$; $\langle 4d \rangle$; $\langle 5d \rangle$; $\langle final\ sign \rangle$
Standard use of the functions `\__fp_basics_pack_low:NNNNNw` and `\__fp_basics_-` `pack_high:NNNNNw`. We finally get to use the $\langle final\ sign \rangle$ which has been sitting there for a while.

```
14045 \cs_new:Npn \__fp_div_significand_small_o:wwwNNNNwN
14046   0 #1; #2; #3; #4#5#6#7#8; #9
14047   {
14048     \exp_after:wN \__fp_basics_pack_high:NNNNNw
14049       \__int_value:w \__int_eval:w 1 #1#2
```

```
14050          \exp_after:wN \__fp_basics_pack_low:NNNNNw
14051          \__int_value:w \__int_eval:w 1 #3#4#5#6#7
14052            + \__fp_round:NNN #9 #7 #8
14053          \exp_after:wN ;
14054      }
```

(*End definition for* `\__fp_div_significand_small_o:wwwNNNNwN.`)

__fp_div_significand_large_o:wwwNNNNwN

$\__fp_div_significand_large_o:wwwNNNNwN \langle 5d \rangle$; $\langle 4d \rangle$; $\langle 4d \rangle$; $\langle 5d \rangle$; $\langle sign \rangle$

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 $\langle rounding\ digit \rangle$ from the last two of our 18 digits.

```
14055 \cs_new:Npn \__fp_div_significand_large_o:wwwNNNNwN
14056      #1; #2; #3; #4#5#6#7#8; #9
14057   {
14058     + \c_one
14059     \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw
14060     \__int_value:w \__int_eval:w 1 #1 #2
14061       \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
14062       \__int_value:w \__int_eval:w 1 #3 #4 #5 #6 +
14063         \exp_after:wN \__fp_round:NNN
14064         \exp_after:wN #9
14065         \exp_after:wN #6
14066         \__int_value:w \__fp_round_digit:Nw #7 #8 ;
14067       \exp_after:wN ;
14068   }
```

(*End definition for* `\__fp_div_significand_large_o:wwwNNNNwN.`)

27.5 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.

```
14069 \cs_new:Npn \__fp_sqrt_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
14070   {
14071     \if_meaning:w 0 #2 \__fp_case_return_same_o:w \fi:
14072     \if_meaning:w 2 #3
14073       \__fp_case_use:nw { \__fp_invalid_operation_o:nw { sqrt } }
14074     \fi:
14075     \if_meaning:w 1 #2 \else: \__fp_case_return_same_o:w \fi:
14076     \__fp_sqrt_npos_o:w
14077     \s__fp \__fp_chk:w #2 #3 #4;
14078   }
```

(*End 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 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;`
14080 `  {`
14081 `    \exp_after:wN \__fp_sanitize:Nw`
14082 `    \exp_after:wN 0`
14083 `    \__int_value:w \__int_eval:w`
14084 `      \if_int_odd:w #1 \exp_stop_f:`
14085 `        \exp_after:wN \__fp_sqrt_npos_auxi_o:wwnnN`
14086 `      \fi:`
14087 `      #1 / \c_two`
14088 `      \__fp_sqrt_Newton_o:wwn 56234133; 0; {#2#3} {#4#5} 0`
14089 `  }`
14090 `\cs_new:Npn \__fp_sqrt_npos_auxi_o:wwnnN #1 / \c_two #2; 0; #3#4#5`
14091 `  {`
14092 `    ( #1 + \c_one ) / \c_two`
14093 `    \__fp_pack_eight:wNNNNNNNN`
14094 `    \__fp_sqrt_npos_auxii_o:wNNNNNNNN`
14095 `    ;`
14096 `    0 #3 #4`
14097 `  }`
14098 `\cs_new:Npn \__fp_sqrt_npos_auxii_o:wNNNNNNNN #1; #2#3#4#5#6#7#8#9`
14099 `  { \__fp_sqrt_Newton_o:wwn 17782794; 0; {#1} {#2#3#4#5#6#7#8#9} }`

(*End 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, $b/c - 1/2 + 1/c \leq [b/c] \leq b/c + 1/2$ and when c is odd, $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2 - 1/(2c)$. For all c, $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2$.

Let us prove that the method converges when implemented with ε-TeX integer division, for any $10^6 \leq a_1 < 10^8$ and starting value $10^6 \leq x < 10^8$. Using the inequalities above and the arithmetic–geometric inequality $(x+t)/2 \geq \sqrt{xt}$ for $t = 10^8 a_1/x$, we find

$$x' = \left[\frac{x + [10^8 a_1/x]}{2}\right] \geq \frac{x + 10^8 a_1/x - 1/2 + 1/(2x)}{2} \geq \sqrt{10^8 a_1} - \frac{1}{4} + \frac{1}{4x}\,.$$

After any step of iteration, we thus have $\delta = x - \sqrt{10^8 a_1} \geq -0.25 + 0.25 \cdot 10^{-8}$. The new difference $\delta' = x' - \sqrt{10^8 a_1}$ after one step is bounded above as

$$x' - \sqrt{10^8 a_1} \leq \frac{x + 10^8 a_1/x + 1/2}{2} + \frac{1}{2} - \sqrt{10^8 a_1} \leq \frac{\delta}{2}\frac{\delta}{\sqrt{10^8 a_1} + \delta} + \frac{3}{4}\,.$$

For $\delta > 3/2$, this last expression is $\leq \delta/2 + 3/4 < \delta$, hence δ decreases at each step: since all x are integers, δ must reach a value $-1/4 < \delta \leq 3/2$. In this range of values, we get $\delta' \leq \frac{3}{4}\frac{3}{2\sqrt{10^8 a_1}} + \frac{3}{4} \leq 0.75 + 1.125 \cdot 10^{-7}$. 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

$$x + [10^8 a_1/x] \leq 2x - 2$$

hence $10^8 a_1/x \leq x - 3/2$, while the second implies

$$x - 1 + [10^8 a_1/(x-1)] \geq 2x - 1$$

hence $10^8 a_1/(x-1) \geq x - 1/2$. Combining the two inequalities yields $x^2 - 3x/2 \geq 10^8 a_1 \geq x - 3x/2 + 1/2$, 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 $[10^7, 10^8]$.

```
14100  \cs_new:Npn \__fp_sqrt_Newton_o:wwn #1; #2; #3
14101    {
14102      \if_int_compare:w #1 = #2 \exp_stop_f:
14103        \exp_after:wN \__fp_sqrt_auxi_o:NNNNwnnN
14104        \__int_value:w \__int_eval:w 9999 9999 +
14105          \exp_after:wN \__fp_use_none_until_s:w
14106      \fi:
14107      \exp_after:wN \__fp_sqrt_Newton_o:wwn
14108      \__int_value:w \__int_eval:w (#1 + #3 * 1 0000 0000 / #1) / \c_two ;
14109      #1; {#3}
14110    }
```

(*End 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` will be 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, $y < 1$.

```
14111  \cs_new:Npn \__fp_sqrt_auxi_o:NNNNwnnN 1 #1#2#3#4#5;
14112    {
14113      \__fp_sqrt_auxii_o:NnnnnnnnN
14114        \__fp_sqrt_auxiii_o:wnnnnnnnn
14115        {#1#2#3#4} {#5} {2499} {9988} {7500}
14116    }
```

(*End 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)

$$10^{4j}z = \left\lceil \left(\lfloor 10^{4j}(a - y^2) \rfloor - 257 \right) \cdot (0.5 \cdot 10^8) \,\middle/\, \lfloor 10^8 y + 1 \rfloor \right\rceil.$$

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

$$10^{4j}(\sqrt{a} - y) = \frac{10^{4j}\left(a - y^2 - (\sqrt{a} - y)^2\right)}{2y} \geq \frac{\lfloor 10^{4j}(a - y^2) \rfloor - 257}{2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor} + \frac{1}{2}$$

where we have replaced the bound $10^{4j}(16 \cdot 10^{-2j}) = 256$ by 257 and extracted the corresponding term $1/\left(2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor\right) \geq 1/2$. 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.

```
14117  \cs_new:Npn \__fp_sqrt_auxii_o:NnnnnnnnN #1 #2#3#4#5#6 #7#8#9
14118    {
14119      \exp_after:wN #1
14120      \__int_value:w \__int_eval:w \c__fp_big_leading_shift_int
14121        + #7 - #2 * #2
14122      \exp_after:wN \__fp_pack_big:NNNNNNw
14123      \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
14124        - 2 * #2 * #3
14125      \exp_after:wN \__fp_pack_big:NNNNNNw
14126      \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
14127        + #8 - #3 * #3 - 2 * #2 * #4
14128      \exp_after:wN \__fp_pack_big:NNNNNNw
14129      \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
14130        - 2 * #3 * #4 - 2 * #2 * #5
14131      \exp_after:wN \__fp_pack_big:NNNNNNw
14132      \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
14133        + #9 000 0000 - #4 * #4 - 2 * #3 * #5 - 2 * #2 * #6
14134      \exp_after:wN \__fp_pack_big:NNNNNNw
14135      \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
14136        - 2 * #4 * #5 - 2 * #3 * #6
14137      \exp_after:wN \__fp_pack_big:NNNNNNw
14138      \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
14139        - #5 * #5 - 2 * #4 * #6
14140      \exp_after:wN \__fp_pack_big:NNNNNNw
14141      \__int_value:w \__int_eval:w
14142        \c__fp_big_middle_shift_int
14143        - 2 * #5 * #6
14144      \exp_after:wN \__fp_pack_big:NNNNNNw
14145      \__int_value:w \__int_eval:w
```

```
14146                          \c__fp_big_trailing_shift_int
14147                          - #6 * #6 ;
14148        % (
14149        - 257 ) * 5000 0000 / (#2#3 + 1) + 10 0000 0000 ;
14150        {#2}{#3}{#4}{#5}{#6} {#7}{#8}#9
14151      }
```

(*End definition for* `\__fp_sqrt_auxii_o:NnnnnnnnN`.)

We receive here the difference $a - y^2 = d = \sum_i d_i \cdot 10^{-4i}$, as $\langle d_2 \rangle$; $\{\langle d_3 \rangle\} \dots \{\langle d_{10} \rangle\}$, 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

$$10^{4j} z = \left[\left(\lfloor 10^{4j}(a - y^2) \rfloor - 257 \right) \cdot (0.5 \cdot 10^8) \Big/ \lfloor 10^8 y + 1 \rfloor \right]$$

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

$$10^{4j} z \geq \frac{(20000 - 257)(0.5 \cdot 10^8)}{\lfloor 10^8 y + 1 \rfloor} \geq (20000 - 257) \cdot 0.5 > 0 \,.$$

```
14152 \cs_new:Npn \__fp_sqrt_auxiii_o:wnnnnnnnn
14153      #1; #2#3#4#5#6#7#8#9
14154      {
14155        \if_int_compare:w #1 > \c_one
14156          \exp_after:wN \__fp_sqrt_auxiv_o:NNNNNw
14157          \__int_value:w \__int_eval:w (#1#2 %)
14158        \else:
14159          \if_int_compare:w #1#2 > \c_one
14160            \exp_after:wN \__fp_sqrt_auxv_o:NNNNNw
14161            \__int_value:w \__int_eval:w (#1#2#3 %)
14162          \else:
14163            \if_int_compare:w #1#2#3 > \c_one
14164              \exp_after:wN \__fp_sqrt_auxvi_o:NNNNNw
14165              \__int_value:w \__int_eval:w (#1#2#3#4 %)
14166            \else:
14167              \exp_after:wN \__fp_sqrt_auxvii_o:NNNNNw
14168              \__int_value:w \__int_eval:w (#1#2#3#4#5 %)
14169            \fi:
14170          \fi:
14171        \fi:
14172      }
14173 \cs_new:Npn \__fp_sqrt_auxiv_o:NNNNNw 1#1#2#3#4#5#6;
14174    { \__fp_sqrt_auxviii_o:nnnnnnn {#1#2#3#4#5#6} {00000000} }
14175 \cs_new:Npn \__fp_sqrt_auxv_o:NNNNNw 1#1#2#3#4#5#6;
```

```
14176     { \__fp_sqrt_auxviii_o:nnnnnnn {000#1#2#3#4#5} {#60000} }
14177 \cs_new:Npn \__fp_sqrt_auxvi_o:NNNNNw 1#1#2#3#4#5#6;
14178     { \__fp_sqrt_auxviii_o:nnnnnnn {0000000#1} {#2#3#4#5#6} }
14179 \cs_new:Npn \__fp_sqrt_auxvii_o:NNNNNw 1#1#2#3#4#5#6;
14180     {
14181         \if_int_compare:w #1#2 = \c_zero
14182             \exp_after:wN \__fp_sqrt_auxx_o:Nnnnnnnn
14183         \fi:
14184         \__fp_sqrt_auxviii_o:nnnnnnn {00000000} {000#1#2#3#4#5}
14185     }
```

(*End definition for* __fp_sqrt_auxiii_o:wnnnnnnnn *and others.*)

__fp_sqrt_auxviii_o:nnnnnnn Simply add the two 8-digit blocks of z, aligned to the last four of the five 4-digit blocks
__fp_sqrt_auxix_o:wnwnw of y, then call the `auxii` auxiliary to evaluate $y'^2 = (y + z)^2$.

```
14186 \cs_new:Npn \__fp_sqrt_auxviii_o:nnnnnnn #1#2 #3#4#5#6#7
14187     {
14188         \exp_after:wN \__fp_sqrt_auxix_o:wnwnw
14189         \__int_value:w \__int_eval:w #3
14190             \exp_after:wN \__fp_basics_pack_low:NNNNNw
14191             \__int_value:w \__int_eval:w #1 + 1#4#5
14192                 \exp_after:wN \__fp_basics_pack_low:NNNNNw
14193                 \__int_value:w \__int_eval:w #2 + 1#6#7 ;
14194     }
14195 \cs_new:Npn \__fp_sqrt_auxix_o:wnwnw #1; #2#3; #4#5;
14196     {
14197         \__fp_sqrt_auxii_o:NnnnnnnnN
14198             \__fp_sqrt_auxiii_o:wnnnnnnnn {#1}{#2}{#3}{#4}{#5}
14199     }
```

(*End definition for* __fp_sqrt_auxviii_o:nnnnnnn *and* __fp_sqrt_auxix_o:wnwnw.)

__fp_sqrt_auxx_o:Nnnnnnnn At this stage, $j = 6$ and $10^{24}z < 10^7$, hence
__fp_sqrt_auxxi_o:wwnnN

$$10^7 + 1/2 > 10^{24}z + 1/2 \geq \left(10^{24}(a - y^2) - 258\right) \cdot (0.5 \cdot 10^8) \Big/ (10^8 y + 1),$$

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 $a \leq 1 - 10^{-16}$.

```
14200 \cs_new:Npn \__fp_sqrt_auxx_o:Nnnnnnnn #1#2#3 #4#5#6#7#8
14201     {
```

```
14202        \exp_after:wN \__fp_sqrt_auxxi_o:wwnnN
14203        \__int_value:w \__int_eval:w
14204          (#8 + 2499) / 5000 * 5000 ;
14205          {#4} {#5} {#6} {#7} ;
14206      }
14207    \cs_new:Npn \__fp_sqrt_auxxi_o:wwnnN #1; #2; #3#4#5
14208      {
14209        \__fp_sqrt_auxii_o:NnnnnnnnN
14210          \__fp_sqrt_auxxii_o:nnnnnnnnw
14211          #2 {#1}
14212          {#3} { #4 + \c_one } #5
14213      }
```

(*End 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.

```
14214    \cs_new:Npn \__fp_sqrt_auxxii_o:nnnnnnnnw 0; #1#2#3#4#5#6#7#8 #9;
14215      {
14216        \if_int_compare:w #1#2 > \c_zero
14217          \if_int_compare:w #1#2 = \c_one
14218            \if_int_compare:w #3#4 = \c_zero
14219              \if_int_compare:w #5#6 = \c_zero
14220                \if_int_compare:w #7#8 = \c_zero
14221                  \__fp_sqrt_auxxiii_o:w
14222                \fi:
14223              \fi:
14224            \fi:
14225          \fi:
14226          \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnN
14227          \__int_value:w 9998
14228        \else:
14229          \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnN
14230          \__int_value:w 10000
14231        \fi:
14232        ;
14233      }
14234    \cs_new:Npn \__fp_sqrt_auxxiii_o:w \fi: \fi: \fi: \fi: #1 \fi: ;
14235      {
14236        \fi: \fi: \fi: \fi: \fi:
14237        \__fp_sqrt_auxxiv_o:wnnnnnnnN 9999 ;
14238      }
```

(*End 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).

647

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.

```
14239 \cs_new:Npn \__fp_sqrt_auxxiv_o:wnnnnnnnN #1; #2#3#4#5#6 #7#8#9
14240   {
14241     \exp_after:wN \__fp_basics_pack_high:NNNNNw
14242     \__int_value:w \__int_eval:w 1 0000 0000 + #2#3
14243       \exp_after:wN \__fp_basics_pack_low:NNNNNw
14244       \__int_value:w \__int_eval:w 1 0000 0000
14245         + #4#5
14246       \if_int_compare:w #6 > #1 \exp_stop_f: + \c_one \fi:
14247       + \exp_after:wN \__fp_round:NNN
14248         \exp_after:wN 0
14249         \exp_after:wN 0
14250         \__int_value:w
14251           \exp_after:wN \use_i:nn
14252           \exp_after:wN \__fp_round_digit:Nw
14253           \__int_value:w \__int_eval:w #6 + 19999 - #1 ;
14254     \exp_after:wN ;
14255   }
```

(*End definition for* `\__fp_sqrt_auxxiv_o:wnnnnnnnN`.)

27.6 Setting the sign

`\__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`.

```
14256 \cs_new:Npn \__fp_set_sign_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
14257   {
14258     \exp_after:wN \__fp_exp_after_o:w
14259     \exp_after:wN \s__fp
14260     \exp_after:wN \__fp_chk:w
14261     \exp_after:wN #2
14262     \__int_value:w
14263       \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
14264     #4;
14265   }
```

(*End definition for* `\__fp_set_sign_o:w`.)

```
14266 ⟨/initex | package⟩
```

28   l3fp-extended implementation

```
14267 ⟨*initex | package⟩
```

⟨@@=fp⟩

28.1 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

$$\{\langle a_1\rangle\} \ \{\langle a_2\rangle\} \ \{\langle a_3\rangle\} \ \{\langle a_4\rangle\} \ \{\langle a_5\rangle\} \ \{\langle a_6\rangle\} \ ;$$

where each $\langle a_i\rangle$ is exactly 4 digits (ranging from 0000 to 9999), except $\langle a_1\rangle$, 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 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 ⟨X_1⟩ ; ⟨X_2⟩ ;
> __fp_fixed_mul:wwn ⟨X_3⟩ ;
> __fp_fixed_add:wwn ⟨X_4⟩ ;

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: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.

28.2 Helpers for numbers with extended precision

\c__fp_one_fixed_tl   The fixed-point number 1, used in l3fp-expo.

```
14269 \tl_const:Nn \c__fp_one_fixed_tl
14270   { {10000} {0000} {0000} {0000} {0000} {0000} }
```

(*End definition for* \c__fp_one_fixed_tl.)

__fp_fixed_continue:wn   This function does nothing. Of course, there is no bound on a_1 (except TeX's own $2^{31}-1$).

```
14271 \cs_new:Npn \__fp_fixed_continue:wn #1; #2 { #2 #1; }
```

(*End definition for* __fp_fixed_continue:wn.)

`\__fp_fixed_add_one:wN` This function adds 1 to the fixed point $\langle a \rangle$, by changing a_1 to $10000 + a_1$, then calls the $\langle continuation \rangle$. This requires $a_1 \leq 2^{31} - 10001$.

```
14272 \cs_new:Npn \__fp_fixed_add_one:wN #1#2; #3
14273   {
14274     \exp_after:wN #3 \exp_after:wN
14275       { \__int_value:w \__int_eval:w \c_ten_thousand + #1 } #2 ;
14276   }
```

(*End 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 $[0, 5 \cdot 10^8 - 1]$.

```
14277 \cs_new:Npn \__fp_fixed_div_myriad:wn #1#2#3#4#5#6;
14278   {
14279     \exp_after:wN \__fp_fixed_mul_after:wwn
14280     \__int_value:w \__int_eval:w \c__fp_leading_shift_int
14281       \exp_after:wN \__fp_pack:NNNNNw
14282       \__int_value:w \__int_eval:w \c__fp_trailing_shift_int
14283         + #1 ; {#2}{#3}{#4}{#5};
14284   }
```

(*End 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 $\langle continuation \rangle$ #2 in front. The $\langle continuation \rangle$ was brought up through the expansions by the packing functions.

```
14285 \cs_new:Npn \__fp_fixed_mul_after:wwn #1; #2; #3 { #3 {#1} #2; }
```

(*End definition for* `\__fp_fixed_mul_after:wwn`.)

28.3 Multiplying a fixed point number by a short one

`\__fp_fixed_mul_short:wwn` 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 $\langle continuation \rangle$ $\{\langle c_1 \rangle\} \ldots \{\langle c_6 \rangle\}$; in the input stream, where each of the $\langle c_i \rangle$ are blocks of 4 digits, except $\langle c_1 \rangle$, which is any TeX integer. Note that indices for $\langle b \rangle$ start at 0: a second operand of {0001}{0000}{0000} will leave the first operand unchanged (rather than dividing it by 10^4, as `\__fp_fixed_mul:wwn` would).

```
14286 \cs_new:Npn \__fp_fixed_mul_short:wwn #1#2#3#4#5#6; #7#8#9;
14287   {
14288     \exp_after:wN \__fp_fixed_mul_after:wwn
14289     \__int_value:w \__int_eval:w \c__fp_leading_shift_int
14290       + #1*#7
14291       \exp_after:wN \__fp_pack:NNNNNw
14292       \__int_value:w \__int_eval:w \c__fp_middle_shift_int
14293         + #1*#8 + #2*#7
14294       \exp_after:wN \__fp_pack:NNNNNw
14295       \__int_value:w \__int_eval:w \c__fp_middle_shift_int
14296         + #1*#9 + #2*#8 + #3*#7
14297         \exp_after:wN \__fp_pack:NNNNNw
14298         \__int_value:w \__int_eval:w \c__fp_middle_shift_int
```

```
14299                    + #2*#9 + #3*#8 + #4*#7
14300                  \exp_after:wN \__fp_pack:NNNNNw
14301                  \__int_value:w \__int_eval:w \c__fp_middle_shift_int
14302                    + #3*#9 + #4*#8 + #5*#7
14303                  \exp_after:wN \__fp_pack:NNNNNw
14304                  \__int_value:w \__int_eval:w \c__fp_trailing_shift_int
14305                    + #4*#9 + #5*#8 + #6*#7
14306                    + ( #5*#9 + #6*#8 + #6*#9 / \c_ten_thousand )
14307                    / \c_ten_thousand ; ;
14308     }
```

(*End definition for* `\__fp_fixed_mul_short:wwn`*.*)

28.4   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`

Divides the fixed point number $\langle a \rangle$ by the (small) integer $0 < \langle n \rangle < 10^4$ and feeds the result to the $\langle continuation \rangle$. 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 will have 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` $\langle continuation \rangle$
> $-1 + Q_1$
> `\__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 ; {$\langle n \rangle$} {$\langle a_6 \rangle$}

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 $\langle continuation \rangle$ as appropriate.

```
14309 \cs_new:Npn \__fp_fixed_div_int:wwN #1#2#3#4#5#6 ; #7 ; #8
14310   {
14311     \exp_after:wN \__fp_fixed_div_int_after:Nw
14312     \exp_after:wN #8
14313     \__int_value:w \__int_eval:w \c_minus_one
14314       \__fp_fixed_div_int:wnN
14315     #1; {#7} \__fp_fixed_div_int_auxi:wnn
14316     #2; {#7} \__fp_fixed_div_int_auxi:wnn
14317     #3; {#7} \__fp_fixed_div_int_auxi:wnn
14318     #4; {#7} \__fp_fixed_div_int_auxi:wnn
14319     #5; {#7} \__fp_fixed_div_int_auxi:wnn
14320     #6; {#7} \__fp_fixed_div_int_auxii:wnn ;
```

```
14321      }
14322  \cs_new:Npn \__fp_fixed_div_int:wnN #1; #2 #3
14323    {
14324      \exp_after:wN #3
14325      \__int_value:w \__int_eval:w #1 / #2 - \c_one ;
14326      {#2}
14327      {#1}
14328    }
14329  \cs_new:Npn \__fp_fixed_div_int_auxi:wnn #1; #2 #3
14330    {
14331      + #1
14332      \exp_after:wN \__fp_fixed_div_int_pack:Nw
14333      \__int_value:w \__int_eval:w 9999
14334        \exp_after:wN \__fp_fixed_div_int:wnN
14335        \__int_value:w \__int_eval:w #3 - #1*#2 \__int_eval_end:
14336    }
14337  \cs_new:Npn \__fp_fixed_div_int_auxii:wnn #1; #2 #3 { + #1 + \c_two ; }
14338  \cs_new:Npn \__fp_fixed_div_int_pack:Nw #1 #2; { + #1; {#2} }
14339  \cs_new:Npn \__fp_fixed_div_int_after:Nw #1 #2; { #1 {#2} }
```

(*End definition for* `\__fp_fixed_div_int:wwN` *and others.*)

28.5   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`

Computes $a + b$ (resp. $a - b$) and feeds the result to the ⟨*continuation*⟩. This function requires $0 \leq a_1, b_1 \leq 114748$, 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.

```
14340  \cs_new:Npn \__fp_fixed_add:wwn { \__fp_fixed_add:Nnnnnwnn + }
14341  \cs_new:Npn \__fp_fixed_sub:wwn { \__fp_fixed_add:Nnnnnwnn - }
14342  \cs_new:Npn \__fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6; #7#8
14343    {
14344      \exp_after:wN \__fp_fixed_add_after:NNNNNwn
14345      \__int_value:w \__int_eval:w 9 9999 9998 + #2#3 #1 #7#8
14346        \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
14347        \__int_value:w \__int_eval:w 1 9999 9998 + #4#5
14348          \__fp_fixed_add:nnNnnnwn #6 #1
14349    }
14350  \cs_new:Npn \__fp_fixed_add:nnNnnnwn #1#2 #3 #4#5 #6#7 ; #8
14351    {
14352      #3 #4#5
14353      \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
14354      \__int_value:w \__int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
14355    }
14356  \cs_new:Npn \__fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6; #7
14357    { + #1 ; {#7} {#2#3#4#5} {#6} }
14358  \cs_new:Npn \__fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6; #7
14359    { #7 {#1#2#3#4#5} {#6} }
```

(*End definition for* `\__fp_fixed_add:wwn` *and others.*)

28.6 Multiplying fixed points

`\__fp_fixed_mul:wwn`
`\__fp_fixed_mul:nnnnnnnw`

Computes $a \times b$ and feeds the result to $\langle continuation \rangle$. 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

$$
\begin{aligned}
a \times b = & a_1 \cdot b_1 \cdot 10^{-8} \\
& + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
& + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1) \cdot 10^{-16} \\
& + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
& + \Big(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \\
& \quad + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \\
& \quad + a_1 \cdot b_5 + a_5 \cdot b_1 \Big) \cdot 10^{-24} + O(10^{-24}),
\end{aligned}
$$

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 $\langle continuation \rangle$ as arguments. It writes the end of the expression, including the right parenthesis and the denominator of the fraction. The $\langle continuation \rangle$ is finally placed in front of the 6 brace groups by `\__fp_fixed_mul_after:wwn`.

```
14360 \cs_new:Npn \__fp_fixed_mul:wwn #1#2#3#4 #5; #6#7#8#9
14361   {
14362     \exp_after:wN \__fp_fixed_mul_after:wwn
14363     \__int_value:w \__int_eval:w \c__fp_leading_shift_int
14364       \exp_after:wN \__fp_pack:NNNNNw
14365       \__int_value:w \__int_eval:w \c__fp_middle_shift_int
14366         + #1*#6
14367       \exp_after:wN \__fp_pack:NNNNNw
14368       \__int_value:w \__int_eval:w \c__fp_middle_shift_int
14369         + #1*#7 + #2*#6
14370       \exp_after:wN \__fp_pack:NNNNNw
14371       \__int_value:w \__int_eval:w \c__fp_middle_shift_int
14372         + #1*#8 + #2*#7 + #3*#6
14373       \exp_after:wN \__fp_pack:NNNNNw
14374       \__int_value:w \__int_eval:w \c__fp_middle_shift_int
14375         + #1*#9 + #2*#8 + #3*#7 + #4*#6
14376       \exp_after:wN \__fp_pack:NNNNNw
14377       \__int_value:w \__int_eval:w \c__fp_trailing_shift_int
14378         + #2*#9 + #3*#8 + #4*#7
14379         + ( #3*#9 + #4*#8
14380           + \__fp_fixed_mul:nnnnnnnw #5 {#6}{#7}   {#1}{#2}
14381   }
```

```
14382 \cs_new:Npn \__fp_fixed_mul:nnnnnnnw #1#2 #3#4 #5#6 #7#8 ;
14383   {
14384     #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c_ten_thousand
14385     + #1*#3 + #5*#7 ; ;
14386   }
```

(*End definition for* `\__fp_fixed_mul:wwn` *and* `\__fp_fixed_mul:nnnnnnnw`.)

28.7   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`

Compute $a \times b + c$, $c - a \times b$, and $1 - a \times b$ and feed the result to the $\langle continuation \rangle$. 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 will perform carries in

$$
\begin{aligned}
a \times b + c = &(a_1 \cdot b_1 + c_1 c_2) \cdot 10^{-8} \\
&+ (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
&+ (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4) \cdot 10^{-16} \\
&+ (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
&+ \Big(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \\
&\quad + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \\
&\quad + a_1 \cdot b_5 + a_5 \cdot b_1 + c_5 c_6 \Big) \cdot 10^{-24} + O(10^{-24}),
\end{aligned}
$$

where $c_1 c_2$, $c_3 c_4$, $c_5 c_6$ 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 $c_1 c_2$ in the first level, calls the `i` auxiliary with arguments described later, and adds a trailing $+ c_5 c_6$; `{`$\langle continuation \rangle$`}` ;. The $+ c_5 c_6$ piece, which is omitted for `\__fp_fixed_one_minus_mul:wwn`, will be taken in the integer expression for the 10^{-24} level.

```
14387 \cs_new:Npn \__fp_fixed_mul_add:wwwn #1; #2; #3#4#5#6#7#8;
14388   {
14389     \exp_after:wN \__fp_fixed_mul_after:wwn
14390     \__int_value:w \__int_eval:w \c__fp_big_leading_shift_int
14391       \exp_after:wN \__fp_pack_big:NNNNNNw
14392       \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int + #3 #4
14393         \__fp_fixed_mul_add:Nwnnwnnn +
14394           + #5 #6 ; #2 ; #1 ; #2 ; +
14395           + #7 #8 ; ;
14396   }
14397 \cs_new:Npn \__fp_fixed_mul_sub_back:wwwn #1; #2; #3#4#5#6#7#8;
14398   {
14399     \exp_after:wN \__fp_fixed_mul_after:wwn
14400     \__int_value:w \__int_eval:w \c__fp_big_leading_shift_int
14401       \exp_after:wN \__fp_pack_big:NNNNNNw
14402       \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int + #3 #4
```

```
14403                \__fp_fixed_mul_add:Nwnnnwnnn -
14404                  + #5 #6 ; #2 ; #1 ; #2 ; -
14405                  + #7 #8 ; ;
14406            }
14407    \cs_new:Npn \__fp_fixed_one_minus_mul:wwn #1; #2;
14408        {
14409            \exp_after:wN \__fp_fixed_mul_after:wwn
14410            \__int_value:w \__int_eval:w \c__fp_big_leading_shift_int
14411                \exp_after:wN \__fp_pack_big:NNNNNNw
14412                \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int + 1 0000 0000
14413                    \__fp_fixed_mul_add:Nwnnnwnnn -
14414                        ; #2 ; #1 ; #2 ; -
14415                        ; ;
14416        }
```

(*End 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` 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.

```
14417    \cs_new:Npn \__fp_fixed_mul_add:Nwnnnwnnn #1 #2; #3#4#5#6; #7#8#9
14418        {
14419            #1 #7*#3
14420            \exp_after:wN \__fp_pack_big:NNNNNNw
14421            \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
14422                #1 #7*#4 #1 #8*#3
14423                \exp_after:wN \__fp_pack_big:NNNNNNw
14424                \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
14425                    #1 #7*#5 #1 #8*#4 #1 #9*#3 #2
14426                    \exp_after:wN \__fp_pack_big:NNNNNNw
14427                    \__int_value:w \__int_eval:w \c__fp_big_middle_shift_int
14428                        #1 \__fp_fixed_mul_add:nnnnwnnnn {#7}{#8}{#9}
14429        }
```

(*End definition for* `\__fp_fixed_mul_add:Nwnnnwnnn`.)

`\__fp_fixed_mul_add:nnnnwnnnn` 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 will 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⟩.

```
14430    \cs_new:Npn \__fp_fixed_mul_add:nnnnwnnnn #1#2#3#4#5; #6#7#8#9
14431        {
14432            ( #1*#9 + #2*#8 + #3*#7 + #4*#6 )
14433            \exp_after:wN \__fp_pack_big:NNNNNNw
```

```
14434        \__int_value:w \__int_eval:w \c__fp_big_trailing_shift_int
14435          \__fp_fixed_mul_add:nnnnwnnwN
14436            { #6 + #4*#7 + #3*#8 + #2*#9 + #1 }
14437            { #7 + #4*#8 + #3*#9 + #2 }
14438            {#1} #5;
14439            {#6}
14440      }
```

(*End definition for* `\__fp_fixed_mul_add:nnnnwnnnn.`)

`\__fp_fixed_mul_add:nnnnwnnwN` Complete the $\langle partial_1 \rangle$ and $\langle partial_2 \rangle$ expressions as explained for the `ii` auxiliary. The second one is divided by 10000: this is the carry from level 10^{-28}. The trailing $+ c_5 c_6$ 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.

```
14441  \cs_new:Npn \__fp_fixed_mul_add:nnnnwnnwN #1#2 #3#4#5; #6#7#8; #9
14442    {
14443      #9 (#4* #1 *#7)
14444      #9 (#5*#6+#4* #2 *#7+#3*#8) / \c_ten_thousand
14445    }
```

(*End definition for* `\__fp_fixed_mul_add:nnnnwnnwN.`)

28.8   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.

`\__fp_ep_to_fixed:wwn` Converts an extended-precision number with an exponent at most 4 to a fixed point
`\__fp_ep_to_fixed_auxi:www` number whose first block will have 12 digits, most often starting with many zeros.
`\__fp_ep_to_fixed_auxii:nnnnnnnwn`

```
14446  \cs_new:Npn \__fp_ep_to_fixed:wwn #1,#2
14447    {
14448      \exp_after:wN \__fp_ep_to_fixed_auxi:www
14449      \__int_value:w \__int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
14450      \exp:w \exp_end_continue_f:w
14451      \prg_replicate:nn { \c_four - \int_max:nn {#1} { -32 } } { 0 } ;
14452    }
14453  \cs_new:Npn \__fp_ep_to_fixed_auxi:www 1#1; #2; #3#4#5#6#7;
14454    {
14455      \__fp_pack_eight:wNNNNNNNN
14456      \__fp_pack_twice_four:wNNNNNNNN
14457      \__fp_pack_twice_four:wNNNNNNNN
14458      \__fp_pack_twice_four:wNNNNNNNN
14459      \__fp_ep_to_fixed_auxii:nnnnnnnwn ;
14460      #2 #1#3#4#5#6#7 0000 !
14461    }
14462  \cs_new:Npn \__fp_ep_to_fixed_auxii:nnnnnnnwn #1#2#3#4#5#6#7; #8! #9
14463    { #9 {#1#2}{#3}{#4}{#5}{#6}{#7}; }
```

(*End 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).

```
14464 \cs_new:Npn \__fp_ep_to_ep:wwN #1,#2#3#4#5#6#7; #8
14465   {
14466     \exp_after:wN #8
14467     \__int_value:w \__int_eval:w #1 + \c_four
14468       \exp_after:wN \use_i:nn
14469       \exp_after:wN \__fp_ep_to_ep_loop:N
14470       \__int_value:w \__int_eval:w 1 0000 0000 + #2 \__int_eval_end:
14471       #3#4#5#6#7 ; ; !
14472   }
14473 \cs_new:Npn \__fp_ep_to_ep_loop:N #1
14474   {
14475     \if_meaning:w 0 #1
14476       - \c_one
14477     \else:
14478       \__fp_ep_to_ep_end:www #1
14479     \fi:
14480     \__fp_ep_to_ep_loop:N
14481   }
14482 \cs_new:Npn \__fp_ep_to_ep_end:www
14483     #1 \fi: \__fp_ep_to_ep_loop:N #2; #3!
14484   {
14485     \fi:
14486     \if_meaning:w ; #1
14487       - \c_two * \c__fp_max_exponent_int
14488       \__fp_ep_to_ep_zero:ww
14489     \fi:
14490     \__fp_pack_twice_four:wNNNNNNNN
14491     \__fp_pack_twice_four:wNNNNNNNN
14492     \__fp_pack_twice_four:wNNNNNNNN
14493     \__fp_use_i:ww , ;
14494     #1 #2 0000 0000 0000 0000 0000 0000 ;
14495   }
14496 \cs_new:Npn \__fp_ep_to_ep_zero:ww \fi: #1; #2; #3;
14497   { \fi: , {1000}{0000}{0000}{0000}{0000}{0000} ; }
```

(*End 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

657

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 $[1000, 9999]$.

```
14498 \cs_new:Npn \__fp_ep_compare:wwww #1,#2#3#4#5#6#7;
14499   { \__fp_ep_compare_aux:wwww {#1}{#2}{#3}{#4}{#5}; #6#7; }
14500 \cs_new:Npn \__fp_ep_compare_aux:wwww #1;#2;#3,#4#5#6#7#8#9;
14501   {
14502     \if_case:w
14503       \__fp_compare_npos:nwnw #1; {#3}{#4}{#5}{#6}{#7}; \exp_stop_f:
14504             \if_int_compare:w #2 = #8#9 \exp_stop_f:
14505               0
14506             \else:
14507               \if_int_compare:w #2 < #8#9 - \fi: 1
14508             \fi:
14509     \or:    1
14510     \else: -1
14511     \fi:
14512   }
```

(*End 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 $[100, 9999]$.

```
14513 \cs_new:Npn \__fp_ep_mul:wwwwn #1,#2; #3,#4;
14514   {
14515     \__fp_ep_to_ep:wwN #3,#4;
14516     \__fp_fixed_continue:wn
14517       {
14518         \__fp_ep_to_ep:wwN #1,#2;
14519         \__fp_ep_mul_raw:wwwwN
14520       }
14521     \__fp_fixed_continue:wn
14522   }
14523 \cs_new:Npn \__fp_ep_mul_raw:wwwwN #1,#2; #3,#4; #5
14524   {
14525     \__fp_fixed_mul:wwn #2; #4;
14526     { \exp_after:wN #5 \__int_value:w \__int_eval:w #1 + #3 , }
14527   }
```

(*End definition for* `\__fp_ep_mul:wwwwn` *and* `\__fp_ep_mul_raw:wwwwN`.)

28.9   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 will first find an integer estimate $a \simeq 10^8/\langle d \rangle$ by computing

$$\alpha = \left[\frac{10^9}{\langle d_1 \rangle + 1}\right]$$

$$\beta = \left[\frac{10^9}{\langle d_1 \rangle}\right]$$

$$a = 10^3 \alpha + (\beta - \alpha) \cdot \left(10^3 - \left[\frac{\langle d_2 \rangle}{10}\right]\right) - 1250,$$

where $\left[\begin{smallmatrix} \bullet \\ \bullet \end{smallmatrix}\right]$ 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 will 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

$$\frac{\langle n \rangle}{\langle d \rangle} = \frac{\langle n \rangle a}{10^8} \left((1 + \epsilon)(1 + \epsilon^2) + \epsilon^4\right).$$

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 $\left[\frac{\langle d_2 \rangle}{10}\right]$ will at most underestimate $10^{-1}(\langle d_2 \rangle + 1)$ by 0.5, as can be checked for each possible last digit of $\langle d_2 \rangle$. Then,

$$10^7 \langle d \rangle a < \left(10^3 \langle d_1 \rangle + \left[\frac{\langle d_2 \rangle}{10}\right] + \frac{1}{2}\right) \left(\left(10^3 - \left[\frac{\langle d_2 \rangle}{10}\right]\right) \beta + \left[\frac{\langle d_2 \rangle}{10}\right] \alpha - 1250\right) \quad (1)$$

$$< \left(10^3 \langle d_1 \rangle + \left[\frac{\langle d_2 \rangle}{10}\right] + \frac{1}{2}\right) \quad (2)$$

$$\left(\left(10^3 - \left[\frac{\langle d_2 \rangle}{10}\right]\right) \left(\frac{10^9}{\langle d_1 \rangle} + \frac{1}{2}\right) + \left[\frac{\langle d_2 \rangle}{10}\right] \left(\frac{10^9}{\langle d_1 \rangle + 1} + \frac{1}{2}\right) - 1250\right) \quad (3)$$

$$< \left(10^3 \langle d_1 \rangle + \left[\frac{\langle d_2 \rangle}{10}\right] + \frac{1}{2}\right) \left(\frac{10^{12}}{\langle d_1 \rangle} - \left[\frac{\langle d_2 \rangle}{10}\right] \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 750\right) \quad (4)$$

We recognize a quadratic polynomial in $[\langle d_2 \rangle/10]$ with a negative leading coefficient: this polynomial is bounded above, according to $([\langle d_2 \rangle/10] + a)(b - c[\langle d_2 \rangle/10]) \le (b + ca)^2/(4c)$. 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 will give us a precise enough answer. Incidentally, the upper bound that we derived tells us that $a < 10^8/\langle d \rangle \le 10^9$, 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.

```
14528 \cs_new:Npn \__fp_ep_div:wwwwn #1,#2; #3,#4;
14529   {
14530     \__fp_ep_to_ep:wwN #1,#2;
14531     \__fp_fixed_continue:wn
14532       {
14533         \__fp_ep_to_ep:wwN #3,#4;
14534         \__fp_ep_div_esti:wwwwn
14535       }
14536   }
```

(*End 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 will compute ⟨*n*⟩/(10⟨*d*⟩)). 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:wnNNNNn`, which computes $10^{-9}a/(1 - \epsilon)$, that is, $1/(10\langle d \rangle)$ and we finally multiply this by the numerator #8.

```
14537 \cs_new:Npn \__fp_ep_div_esti:wwwwn #1,#2#3; #4,
14538   {
14539     \exp_after:wN \__fp_ep_div_estii:wwnnwwn
14540     \__int_value:w \__int_eval:w 10 0000 0000 / ( #2 + \c_one )
14541       \exp_after:wN ;
14542     \__int_value:w \__int_eval:w #4 - #1 + \c_one ,
14543     {#2} #3;
14544   }
14545 \cs_new:Npn \__fp_ep_div_estii:wwnnwwn #1; #2,#3#4#5; #6; #7
14546   {
```

```
14547        \exp_after:wN \__fp_ep_div_estiii:NNNNNwwwn
14548        \__int_value:w \__int_eval:w 10 0000 0000 - 1750
14549          + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
14550        {#3}{#4}#5; #6; { #7 #2, }
14551      }
14552    \cs_new:Npn \__fp_ep_div_estiii:NNNNNwwwn 1#1#2#3#4#5#6; #7;
14553      {
14554        \__fp_fixed_mul_short:wwn #7; {#1}{#2#3#4#5}{#6};
14555        \__fp_ep_div_epsi:wnNNNNNn {#1#2#3#4}#5#6
14556        \__fp_fixed_mul:wwn
14557      }
```

(*End 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`.

```
14558    \cs_new:Npn \__fp_ep_div_epsi:wnNNNNNn #1#2#3#4#5#6;
14559      {
14560        \exp_after:wN \__fp_ep_div_epsii:wwnNNNNNn
14561        \__int_value:w \__int_eval:w 1 9998 - #2
14562          \exp_after:wN \__fp_ep_div_eps_pack:NNNNNw
14563          \__int_value:w \__int_eval:w 1 9999 9998 - #3#4
14564            \exp_after:wN \__fp_ep_div_eps_pack:NNNNNw
14565            \__int_value:w \__int_eval:w 2 0000 0000 - #5#6 ; ;
14566      }
14567    \cs_new:Npn \__fp_ep_div_eps_pack:NNNNNw #1#2#3#4#5#6;
14568      { + #1 ; {#2#3#4#5} {#6} }
14569    \cs_new:Npn \__fp_ep_div_epsii:wwnNNNNNn 1#1; #2; #3#4#5#6#7#8
14570      {
14571        \__fp_fixed_mul:wwn {0000}{#1}#2; {0000}{#1}#2;
14572        \__fp_fixed_add_one:wN
14573        \__fp_fixed_mul:wwn {10000} {#1} #2 ;
14574        {
14575          \__fp_fixed_mul_short:wwn {0000}{#1}#2; {#3}{#4#5#6#7}{#8000};
14576          \__fp_fixed_div_myriad:wn
14577          \__fp_fixed_mul:wwn
14578        }
14579        \__fp_fixed_add:wwn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000};
14580      }
```

(*End definition for* __fp_ep_div_epsi:wnNNNNNn, __fp_ep_div_eps_pack:NNNNNw, *and* __fp_ep_-
div_epsii:wwnNNNNNn.)

28.10   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 will replace 10^8 by a slightly larger number which will ensure 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.

```
14581 \cs_new:Npn \__fp_ep_isqrt:wwn #1,#2;
14582   {
14583     \__fp_ep_to_ep:wwN #1,#2;
14584     \__fp_ep_isqrt_auxi:wwn
14585   }
14586 \cs_new:Npn \__fp_ep_isqrt_auxi:wwn #1,
14587   {
14588     \exp_after:wN \__fp_ep_isqrt_auxii:wwnnnwn
14589     \__int_value:w \__int_eval:w
14590       \int_if_odd:nTF {#1}
14591         { (\c_one - #1) / \c_two , 535 , { 0 } { } }
14592         { \c_one - #1 / \c_two , 168 , { } { 0 } }
14593   }
14594 \cs_new:Npn \__fp_ep_isqrt_auxii:wwnnnwn #1, #2, #3#4 #5#6; #7
14595   {
14596     \__fp_ep_isqrt_esti:wwwnnwn #2, 0, #5, {#3} {#4}
14597       {#5} #6 ; { #7 #1 , }
14598   }
```

(*End 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:NNNNNwwn

If the last two approximations gave the same result, we are done: call the `estii` function to clean up. Otherwise, evaluate $(\langle prev \rangle + 1.005 \cdot 10^{8 \text{ or } 9}/(\langle prev \rangle \cdot x))/2$, 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 $100 r^2/2$ or $10 r^2/2$ (when the exponent is even or odd, respectively) and feed that to `estiii`. This third auxiliary finds $y_{\text{even}}/2 = 10^{-4} r^2 x/2$ or $y_{\text{odd}}/2 = 10^{-5} r^2 x/2$ (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.

```
14599 \cs_new:Npn \__fp_ep_isqrt_esti:wwwnnwn #1, #2, #3, #4
14600   {
14601     \if_int_compare:w #1 = #2 \exp_stop_f:
14602       \exp_after:wN \__fp_ep_isqrt_estii:wwwnnwn
14603     \fi:
14604     \exp_after:wN \__fp_ep_isqrt_esti:wwwnnwn
14605     \__int_value:w \__int_eval:w
14606       (#1 + 1 0050 0000 #4 / (#1 * #3)) / \c_two ,
14607     #1, #3, {#4}
14608   }
14609 \cs_new:Npn \__fp_ep_isqrt_estii:wwwnnwn #1, #2, #3, #4#5
14610   {
14611     \exp_after:wN \__fp_ep_isqrt_estiii:NNNNNwwwn
14612     \__int_value:w \__int_eval:w 1000 0000 + #2 * #2 #5 * \c_five
14613       \exp_after:wN , \__int_value:w \__int_eval:w 10000 + #2 ;
14614   }
14615 \cs_new:Npn \__fp_ep_isqrt_estiii:NNNNNwwwn 1#1#2#3#4#5#6, 1#7#8; #9;
14616   {
14617     \__fp_fixed_mul_short:wwn #9; {#1} {#2#3#4#5} {#600} ;
14618     \__fp_ep_isqrt_epsi:wN
14619     \__fp_fixed_mul_short:wwn {#7} {#80} {0000} ;
14620   }
```

(*End 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 $z \mapsto z(3/2 - z^2 y/2)$. 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.

```
14621 \cs_new:Npn \__fp_ep_isqrt_epsi:wN #1;
14622   {
14623     \__fp_fixed_sub:wwn {15000}{0000}{0000}{0000}{0000}{0000}; #1;
14624     \__fp_ep_isqrt_epsii:wwN #1;
14625     \__fp_ep_isqrt_epsii:wwN #1;
14626     \__fp_ep_isqrt_epsii:wwN #1;
14627   }
14628 \cs_new:Npn \__fp_ep_isqrt_epsii:wwN #1; #2;
14629   {
14630     \__fp_fixed_mul:wwn #1; #1;
14631     \__fp_fixed_mul_sub_back:wwwn #2;
14632       {15000}{0000}{0000}{0000}{0000}{0000};
14633     \__fp_fixed_mul:wwn #1;
14634   }
```

(*End definition for* `\__fp_ep_isqrt_epsi:wN` *and* `\__fp_ep_isqrt_epsii:wwN`*.*)

28.11 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:wwN
__fp_ep_inv_to_float: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.

```
14635 \cs_new:Npn \__fp_ep_to_float:wwN #1,
14636   { + \__int_eval:w #1 \__fp_fixed_to_float:wN }
14637 \cs_new:Npn \__fp_ep_inv_to_float:wwN #1,#2;
14638   {
14639     \__fp_ep_div:wwwwn 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1,#2;
14640     \__fp_ep_to_float:wwN
14641   }
```

(End definition for __fp_ep_to_float:wwN *and* __fp_ep_inv_to_float:wwN.*)*

__fp_fixed_inv_to_float:wN Another function which reduces to converting an extended precision number to a float.

```
14642 \cs_new:Npn \__fp_fixed_inv_to_float:wN
14643   { \__fp_ep_inv_to_float:wwN 0, }
```

(End definition for __fp_fixed_inv_to_float:wN.*)*

__fp_fixed_to_float_rad:wN Converts the fixed point number #1 from degrees to radians then to a floating point number. This could perhaps remain in l3fp-trig.

```
14644 \cs_new:Npn \__fp_fixed_to_float_rad:wN #1;
14645   {
14646     \__fp_fixed_mul:wwn #1; {5729}{5779}{5130}{8232}{0876}{7981};
14647     { \__fp_ep_to_float:wwN 2, }
14648   }
```

(End definition for __fp_fixed_to_float_rad:wN.*)*

__fp_fixed_to_float:wN
__fp_fixed_to_float:Nw

yields

$$\langle exponent'\rangle \; ; \; \{\langle a'_1\rangle\} \; \{\langle a'_2\rangle\} \; \{\langle a'_3\rangle\} \; \{\langle a'_4\rangle\} \; ;$$

And the to_fixed version gives six brace groups instead of 4, ensuring that $1000 \leq \langle a'_1\rangle \leq 9999$. At this stage, we know that $\langle a_1\rangle$ is positive (otherwise, it is sign of an error before), and we assume that it is less than 10^8.[11]

```
14649 \cs_new:Npn \__fp_fixed_to_float:Nw #1#2; { \__fp_fixed_to_float:wN #2; #1 }
14650 \cs_new:Npn \__fp_fixed_to_float:wN #1#2#3#4#5#6; #7
14651   {
14652     + \__int_eval:w \c_four % for the 8-digit-at-the-start thing.
14653     \exp_after:wN \exp_after:wN
14654     \exp_after:wN \__fp_fixed_to_loop:N
14655     \exp_after:wN \use_none:n
14656     \__int_value:w \__int_eval:w
14657       1 0000 0000 + #1   \exp_after:wN \__fp_use_none_stop_f:n
14658       \__int_value:w   1#2 \exp_after:wN \__fp_use_none_stop_f:n
14659       \__int_value:w 1#3#4 \exp_after:wN \__fp_use_none_stop_f:n
14660       \__int_value:w 1#5#6
14661     \exp_after:wN ;
14662     \exp_after:wN ;
14663   }
14664 \cs_new:Npn \__fp_fixed_to_loop:N #1
14665   {
```

---

[11]Bruno: I must double check this assumption.

```
14666        \if_meaning:w 0 #1
14667            - \c_one
14668            \exp_after:wN \__fp_fixed_to_loop:N
14669        \else:
14670            \exp_after:wN \__fp_fixed_to_loop_end:w
14671            \exp_after:wN #1
14672        \fi:
14673    }
14674 \cs_new:Npn \__fp_fixed_to_loop_end:w #1 #2 ;
14675    {
14676        \if_meaning:w ; #1
14677            \exp_after:wN \__fp_fixed_to_float_zero:w
14678        \else:
14679            \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
14680            \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
14681            \exp_after:wN \__fp_fixed_to_float_pack:ww
14682            \exp_after:wN ;
14683        \fi:
14684        #1 #2 0000 0000 0000 0000 ;
14685    }
14686 \cs_new:Npn \__fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
14687    {
14688        - \c_two * \c__fp_max_exponent_int ;
14689        {0000} {0000} {0000} {0000} ;
14690    }
14691 \cs_new:Npn \__fp_fixed_to_float_pack:ww #1 ; #2#3 ; ;
14692    {
14693        \if_int_compare:w #2 > \c_four
14694            \exp_after:wN \__fp_fixed_to_float_round_up:wnnnnw
14695        \fi:
14696        ; #1 ;
14697    }
14698 \cs_new:Npn \__fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
14699    {
14700        \exp_after:wN \__fp_basics_pack_high:NNNNNw
14701        \__int_value:w \__int_eval:w 1 #1#2
14702            \exp_after:wN \__fp_basics_pack_low:NNNNNw
14703            \__int_value:w \__int_eval:w 1 #3#4 + \c_one ;
14704    }
```

(*End definition for* __fp_fixed_to_float:wN *and* __fp_fixed_to_float:Nw.)

```
14705 ⟨/initex | package⟩
```

29 **l3fp-expo** implementation

```
14706 ⟨*initex | package⟩
14707 ⟨@@=fp⟩
```

29.1 Logarithm

29.1.1 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, $c \in [1, 10]$ will be 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

$$t = \frac{\epsilon}{1 - \epsilon} = \epsilon(1 + \epsilon)(1 + \epsilon^2)(1 + \epsilon^4)\dots,$$

is not too difficult to compute.

29.1.2 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 $\ln(5)$.

```
14708 \tl_const:Nn \c__fp_ln_i_fixed_tl    { {0000}{0000}{0000}{0000}{0000}{0000} }
14709 \tl_const:Nn \c__fp_ln_ii_fixed_tl   { {6931}{4718}{0559}{9453}{0941}{7232} }
14710 \tl_const:Nn \c__fp_ln_iii_fixed_tl {{10986}{1228}{8668}{1096}{9139}{5245} }
14711 \tl_const:Nn \c__fp_ln_iv_fixed_tl  {{13862}{9436}{1119}{8906}{1883}{4464} }
14712 \tl_const:Nn \c__fp_ln_vi_fixed_tl  {{17917}{5946}{9228}{0550}{0081}{2477} }
14713 \tl_const:Nn \c__fp_ln_vii_fixed_tl {{19459}{1014}{9055}{3133}{0510}{5353} }
14714 \tl_const:Nn \c__fp_ln_viii_fixed_tl{{20794}{4154}{1679}{8359}{2825}{1696} }
14715 \tl_const:Nn \c__fp_ln_ix_fixed_tl  {{21972}{2457}{7336}{2193}{8279}{0490} }
14716 \tl_const:Nn \c__fp_ln_x_fixed_tl   {{23025}{8509}{2994}{0456}{8401}{7991} }
```

(*End definition for* \c__fp_ln_i_fixed_tl *and others.*)

29.1.3 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.

```
14717 \cs_new:Npn \__fp_ln_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
14718   {
14719     \if_meaning:w 2 #3
14720       \__fp_case_use:nw { \__fp_invalid_operation_o:nw { ln } }
14721     \fi:
14722     \if_case:w #2 \exp_stop_f:
14723       \__fp_case_use:nw
14724         { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
14725     \or:
14726     \else:
```

```
14727          \__fp_case_return_same_o:w
14728        \fi:
14729        \__fp_ln_npos_o:w \s__fp \__fp_chk:w #2#3#4;
14730      }
```

(*End definition for* `\__fp_ln_o:w`.)

29.1.4 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.

```
14731  \cs_new:Npn \__fp_ln_npos_o:w \s__fp \__fp_chk:w 10#1#2#3;
14732    { %^^A todo: ln(1) should be "exact zero", not "underflow"
14733      \exp_after:wN \__fp_sanitize:Nw
14734      \__int_value:w % for the overall sign
14735        \if_int_compare:w #1 < \c_one
14736          2
14737        \else:
14738          0
14739        \fi:
14740      \exp_after:wN \exp_stop_f:
14741      \__int_value:w \__int_eval:w % for the exponent
14742        \__fp_ln_significand:NNNNnnnN #2#3
14743        \__fp_ln_exponent:wn {#1}
14744    }
```

(*End definition for* `\__fp_ln_npos_o:w`.)

`\__fp_ln_significand:NNNNnnnN`   `\__fp_ln_significand:NNNNnnnN` $\langle X_1 \rangle$ {$\langle X_2 \rangle$} {$\langle X_3 \rangle$} {$\langle X_4 \rangle$} $\langle continuation \rangle$
This function expands to

$\langle continuation \rangle$ {$\langle Y_1 \rangle$} {$\langle Y_2 \rangle$} {$\langle Y_3 \rangle$} {$\langle Y_4 \rangle$} {$\langle Y_5 \rangle$} {$\langle Y_6 \rangle$} ;

where $Y = -\ln(X)$ as an extended fixed point.

```
14745  \cs_new:Npn \__fp_ln_significand:NNNNnnnN #1#2#3#4
14746    {
14747      \exp_after:wN \__fp_ln_x_ii:wnnnn
14748      \__int_value:w
14749        \if_case:w #1 \exp_stop_f:
14750        \or:
14751          \if_int_compare:w #2 < \c_four
14752            \__int_eval:w \c_ten - #2
14753          \else:
14754            6
14755          \fi:
14756        \or: 4
14757        \or: 3
14758        \or: 2
14759        \or: 2
14760        \or: 2
14761        \else: 1
14762        \fi:
14763      ; { #1 #2 #3 #4 }
14764    }
```

__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 $1 + x = 1 + ac \in [1.7, 2.4)$.

```
14765 \cs_new:Npn \__fp_ln_x_ii:wnnnn #1; #2#3#4#5
14766   {
14767     \exp_after:wN \__fp_ln_div_after:Nw
14768     \cs:w c__fp_ln_ \__int_to_roman:w #1 _fixed_tl \exp_after:wN \cs_end:
14769     \__int_value:w
14770       \exp_after:wN \__fp_ln_x_iv:wnnnnnnnn
14771       \__int_value:w \__int_eval:w
14772         \exp_after:wN \__fp_ln_x_iii_var:NNNNNw
14773         \__int_value:w \__int_eval:w 9999 9990 + #1*#2#3 +
14774           \exp_after:wN \__fp_ln_x_iii:NNNNNNw
14775           \__int_value:w \__int_eval:w 10 0000 0000 + #1*#4#5 ;
14776       {20000} {0000} {0000} {0000}
14777   } %^^A todo: reoptimize (a generalization attempt failed).
14778 \cs_new:Npn \__fp_ln_x_iii:NNNNNNw #1#2 #3#4#5#6 #7;
14779   { #1#2; {#3#4#5#6} {#7} }
14780 \cs_new:Npn \__fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6;
14781   {
14782     #1#2#3#4#5 + \c_one ;
14783     {#1#2#3#4#5} {#6}
14784   }
```

The Taylor series will be 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 eTeX 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

$$10^4 B \leq A_1 A_2 . A_3 A_4 - \left(\frac{A_1 A_2}{y} - \frac{3}{2} \right) 10^4 Z$$

$$\leq A_1 A_2 \left(1 - \frac{10^4 Z}{y} \right) + 1 + \frac{3}{2} y$$

$$\leq 10^8 \frac{A}{y} + 1 + \frac{3}{2} y$$

In the same way, and using $1.7 \cdot 10^4 \leq y \leq 2.4 \cdot 10^4$, and convexity, we get

$$10^4 A = 2 \cdot 10^4$$

$$10^4 B \leq 10^8 \frac{A}{y} + 1.6y \leq 4.7 \cdot 10^4$$

$$10^4 C \leq 10^8 \frac{B}{y} + 1.6y \leq 5.8 \cdot 10^4$$

$$10^4 D \leq 10^8 \frac{C}{y} + 1.6y \leq 6.3 \cdot 10^4$$

$$10^4 E \leq 10^8 \frac{D}{y} + 1.6y \leq 6.5 \cdot 10^4$$

$$10^4 F \leq 10^8 \frac{E}{y} + 1.6y \leq 6.6 \cdot 10^4$$

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).[12]

$\_\_fp\_ln\_x\_iv:wnnnnnnnn$ $\langle 1 \text{ or } 2 \rangle$ $\langle 8d \rangle$; {$\langle 4d \rangle$} {$\langle 4d \rangle$} $\langle \text{fixed-tl} \rangle$

The number is x. Compute y by adding 1 to the five first digits.

```
14785 \cs_new:Npn \__fp_ln_x_iv:wnnnnnnnn #1; #2#3#4#5 #6#7#8#9
14786   {
14787     \exp_after:wN \__fp_div_significand_pack:NNN
14788     \__int_value:w \__int_eval:w
14789     \__fp_ln_div_i:w #1 ;
14790       #6 #7 ; {#8} {#9}
14791       {#2} {#3} {#4} {#5}
14792       { \exp_after:wN \__fp_ln_div_ii:wwn \__int_value:w #1 }
14793       { \exp_after:wN \__fp_ln_div_ii:wwn \__int_value:w #1 }
14794       { \exp_after:wN \__fp_ln_div_ii:wwn \__int_value:w #1 }
14795       { \exp_after:wN \__fp_ln_div_ii:wwn \__int_value:w #1 }
14796       { \exp_after:wN \__fp_ln_div_vi:wwn \__int_value:w #1 }
14797   }
14798 \cs_new:Npn \__fp_ln_div_i:w #1;
14799   {
14800     \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
14801     \__int_value:w \__int_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
14802   }
14803 \cs_new:Npn \__fp_ln_div_ii:wwn #1; #2;#3 % y; B1;B2 <- for k=1
14804   {
14805     \exp_after:wN \__fp_div_significand_pack:NNN
14806     \__int_value:w \__int_eval:w
14807       \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
14808       \__int_value:w \__int_eval:w 999999 + #2 #3 / #1 ; % Q2
14809       #2 #3 ;
14810   }
14811 \cs_new:Npn \__fp_ln_div_vi:wwn #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
14812   {
```

---

[12]Bruno: to be completed.

```
14813        \exp_after:wN \__fp_div_significand_pack:NNN
14814        \__int_value:w \__int_eval:w 1000000 + #2 #3 / #1 ; % Q6
14815      }
```

We now have essentially[13]

> $\texttt{\textbackslash\_\_fp\_ln\_div\_after:Nw}$ $\langle\textit{fixed tl}\rangle$ $\texttt{\textbackslash\_\_fp\_div\_significand\_pack:NNN}$ $10^6 +$
> Q_1 $\texttt{\textbackslash\_\_fp\_div\_significand\_pack:NNN}$ $10^6 + Q_2$ $\texttt{\textbackslash\_\_fp\_div\_significand\_-}$
> $\texttt{pack:NNN}$ $10^6 + Q_3$ $\texttt{\textbackslash\_\_fp\_div\_significand\_pack:NNN}$ $10^6 + Q_4$ $\texttt{\textbackslash\_\_fp\_-}$
> $\texttt{div\_significand\_pack:NNN}$ $10^6 + Q_5$ $\texttt{\textbackslash\_\_fp\_div\_significand\_pack:NNN}$
> $10^6 + Q_6$; $\langle\textit{exponent}\rangle$; $\langle\textit{continuation}\rangle$

where $\langle\textit{fixed tl}\rangle$ holds the logarithm of a number in $[1, 10]$, and $\langle\textit{exponent}\rangle$ is the exponent. Also, the expansion is done backwards. Then $\texttt{\textbackslash\_\_fp\_div\_significand\_pack:NNN}$ 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

> $\texttt{\textbackslash\_\_fp\_ln\_div\_after:Nw}$ $\langle\textit{fixed-tl}\rangle$ $\langle\textit{1d}\rangle$; $\langle\textit{4d}\rangle$; $\langle\textit{4d}\rangle$; $\langle\textit{4d}\rangle$; $\langle\textit{4d}\rangle$;
> $\langle\textit{4d}\rangle$; $\langle\textit{exponent}\rangle$;

Just as with division, we know that the first two digits are $\texttt{1}$ and $\texttt{0}$ 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.

```
14816  \cs_new:Npn \__fp_ln_div_after:Nw #1#2;
14817    {
14818      \if_meaning:w 0 #2
14819        \exp_after:wN \__fp_ln_t_small:Nw
14820      \else:
14821        \exp_after:wN \__fp_ln_t_large:NNw
14822        \exp_after:wN -
14823      \fi:
14824      #1
14825    }
14826  \cs_new:Npn \__fp_ln_t_small:Nw #1 #2; #3; #4; #5; #6; #7;
14827    {
14828      \exp_after:wN \__fp_ln_t_large:NNw
14829      \exp_after:wN + % <sign>
14830      \exp_after:wN #1
14831      \__int_value:w \__int_eval:w 9999 - #2 \exp_after:wN ;
14832      \__int_value:w \__int_eval:w 9999 - #3 \exp_after:wN ;
14833      \__int_value:w \__int_eval:w 9999 - #4 \exp_after:wN ;
14834      \__int_value:w \__int_eval:w 9999 - #5 \exp_after:wN ;
14835      \__int_value:w \__int_eval:w 9999 - #6 \exp_after:wN ;
14836      \__int_value:w \__int_eval:w 1 0000 - #7 ;
14837    }
```

> $\texttt{\textbackslash\_\_fp\_ln\_t\_large:NNw}$ $\langle\textit{sign}\rangle\langle\textit{fixed tl}\rangle$ $\langle t_1\rangle$; $\langle t_2\rangle$; $\langle t_3\rangle$; $\langle t_4\rangle$; $\langle t_5\rangle$; $\langle t_6\rangle$;
> $\langle\textit{exponent}\rangle$; $\langle\textit{continuation}\rangle$

Compute the square $\texttt{t}^2$, and keep $\texttt{t}$ at the end with its sign. We know that $\texttt{t} < 0.1765$, so every piece has at most 4 digits. However, since we were not careful in $\texttt{\textbackslash\_\_fp\_ln\_t\_-}$ $\texttt{small:w}$, they can have less than 4 digits.

---

[13]Bruno: add a mention that the error on Q_6 is bounded by 10 (probably 6.7), and thus corresponds to an error of 10^{-23} on the final result, small enough in all cases.

```
14838 \cs_new:Npn \__fp_ln_t_large:NNw #1 #2 #3; #4; #5; #6; #7; #8;
14839   {
14840     \exp_after:wN \__fp_ln_square_t_after:w
14841     \__int_value:w \__int_eval:w 9999 0000 + #3*#3
14842       \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
14843       \__int_value:w \__int_eval:w 9999 0000 + 2*#3*#4
14844         \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
14845         \__int_value:w \__int_eval:w 9999 0000 + 2*#3*#5 + #4*#4
14846           \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
14847           \__int_value:w \__int_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
14848             \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
14849             \__int_value:w \__int_eval:w 1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
14850               + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
14851               % ; ; ;
14852     \exp_after:wN \__fp_ln_twice_t_after:w
14853     \__int_value:w \__int_eval:w -1 + 2*#3
14854       \exp_after:wN \__fp_ln_twice_t_pack:Nw
14855       \__int_value:w \__int_eval:w 9999 + 2*#4
14856         \exp_after:wN \__fp_ln_twice_t_pack:Nw
14857         \__int_value:w \__int_eval:w 9999 + 2*#5
14858           \exp_after:wN \__fp_ln_twice_t_pack:Nw
14859           \__int_value:w \__int_eval:w 9999 + 2*#6
14860             \exp_after:wN \__fp_ln_twice_t_pack:Nw
14861             \__int_value:w \__int_eval:w 9999 + 2*#7
14862               \exp_after:wN \__fp_ln_twice_t_pack:Nw
14863               \__int_value:w \__int_eval:w 10000 + 2*#8 ; ;
14864     { \__fp_ln_c:NwNw #1 }
14865     #2
14866   }
14867 \cs_new:Npn \__fp_ln_twice_t_pack:Nw #1 #2; { + #1 ; {#2} }
14868 \cs_new:Npn \__fp_ln_twice_t_after:w #1; { ;;; {#1} }
14869 \cs_new:Npn \__fp_ln_square_t_pack:NNNNNw #1 #2#3#4#5 #6;
14870   { + #1#2#3#4#5 ; {#6} }
14871 \cs_new:Npn \__fp_ln_square_t_after:w 1 0 #1#2#3 #4;
14872   { \__fp_ln_Taylor:wwNw {0#1#2#3} {#4} }
```

(*End definition for* __fp_ln_x_ii:wnnnn.)

__fp_ln_Taylor:wwNw Denoting $T = t^2$, we get

> __fp_ln_Taylor:wwNw {⟨T_1⟩} {⟨T_2⟩} {⟨T_3⟩} {⟨T_4⟩} {⟨T_5⟩} {⟨T_6⟩} ; ;
> {⟨$(2t)_1$⟩} {⟨$(2t)_2$⟩} {⟨$(2t)_3$⟩} {⟨$(2t)_4$⟩} {⟨$(2t)_5$⟩} {⟨$(2t)_6$⟩} ; { __fp_ln_-
> c:NwNn ⟨*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;
```

671

This uses the routine for dividing a number by a small integer ($< 10^4$).

```
14873 \cs_new:Npn \__fp_ln_Taylor:wwNw
14874   { \__fp_ln_Taylor_loop:www 21 ; {0000}{0000}{0000}{0000}{0000}{0000} ; }
14875 \cs_new:Npn \__fp_ln_Taylor_loop:www #1; #2; #3;
14876   {
14877     \if_int_compare:w #1 = \c_one
14878       \__fp_ln_Taylor_break:w
14879     \fi:
14880     \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl ; #1;
14881     \__fp_fixed_add:wwn #2;
14882     \__fp_fixed_mul:wwn #3;
14883       {
14884         \exp_after:wN \__fp_ln_Taylor_loop:www
14885         \__int_value:w \__int_eval:w #1 - \c_two ;
14886       }
14887     #3;
14888   }
14889 \cs_new:Npn \__fp_ln_Taylor_break:w \fi: #1 \__fp_fixed_add:wwn #2#3; #4 ;;
14890   {
14891     \fi:
14892     \exp_after:wN \__fp_fixed_mul:wwn
14893     \exp_after:wN { \__int_value:w \__int_eval:w 10000 + #2 } #3;
14894   }
```

(*End definition for* `\__fp_ln_Taylor:wwNw`.)

`\__fp_ln_c:NwNw`　　　　`\__fp_ln_c:NwNw` $\langle sign\rangle$ {$\langle r_1\rangle$} {$\langle r_2\rangle$} {$\langle r_3\rangle$} {$\langle r_4\rangle$} {$\langle r_5\rangle$} {$\langle r_6\rangle$} ; $\langle fixed\ tl\rangle$ $\langle exponent\rangle$; $\langle continuation\rangle$

We are now reduced to finding $\ln(c)$ and $\langle exponent\rangle \ln(10)$ 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 will be at least $\ln(10/7) \simeq 0.35$.[15]

```
14895 \cs_new:Npn \__fp_ln_c:NwNw #1 #2; #3
14896   {
14897     \if_meaning:w + #1
14898       \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_sub:wwn
14899     \else:
14900       \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_add:wwn
14901     \fi:
14902     #3 ; #2 ;
14903   }
```

(*End definition for* `\__fp_ln_c:NwNw`.)

`\__fp_ln_exponent:wn`　　　　`\__fp_ln_exponent:wn` {$\langle s_1\rangle$} {$\langle s_2\rangle$} {$\langle s_3\rangle$} {$\langle s_4\rangle$} {$\langle s_5\rangle$} {$\langle s_6\rangle$} ; {$\langle exponent\rangle$}

---

[14]Bruno: add explanations.

[15]Bruno: that was wrong at some point, I must check.

[16]Bruno: this **must** be updated with correct values!

Compute ⟨*exponent*⟩ times ln(10). Apart from the cases where ⟨*exponent*⟩ is 0 or 1, the result will necessarily be at least ln(10) ≃ 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.

```
14904 \cs_new:Npn \__fp_ln_exponent:wn #1; #2
14905   {
14906     \if_case:w #2 \exp_stop_f:
14907       \c_zero \__fp_case_return:nw { \__fp_fixed_to_float:Nw 2 }
14908     \or:
14909       \exp_after:wN \__fp_ln_exponent_one:ww \__int_value:w
14910     \else:
14911       \if_int_compare:w #2 > \c_zero
14912         \exp_after:wN \__fp_ln_exponent_small:NNww
14913         \exp_after:wN 0
14914         \exp_after:wN \__fp_fixed_sub:wwn \__int_value:w
14915       \else:
14916         \exp_after:wN \__fp_ln_exponent_small:NNww
14917         \exp_after:wN 2
14918         \exp_after:wN \__fp_fixed_add:wwn \__int_value:w -
14919       \fi:
14920     \fi:
14921     #2; #1;
14922   }
```

Now we painfully write all the cases.[17] No overflow nor underflow can happen, except when computing `ln(1)`.

```
14923 \cs_new:Npn \__fp_ln_exponent_one:ww 1; #1;
14924   {
14925     \c_zero
14926     \exp_after:wN \__fp_fixed_sub:wwn \c__fp_ln_x_fixed_tl ; #1;
14927     \__fp_fixed_to_float:wN 0
14928   }
```

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.

```
14929 \cs_new:Npn \__fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
14930   {
14931     \c_four
14932     \exp_after:wN \__fp_fixed_mul:wwn
14933       \c__fp_ln_x_fixed_tl ;
14934       {#3}{0000}{0000}{0000}{0000}{0000} ;
14935     #2
14936       {0000}{#4}{#5}{#6}{#7}{#8};
14937     \__fp_fixed_to_float:wN #1
14938   }
```

(*End definition for* `\__fp_ln_exponent:wn`.)

---

[17]Bruno: do rounding.

29.2 Exponential

29.2.1 Sign, exponent, and special numbers

__fp_exp_o:w

```
14939 \cs_new:Npn \__fp_exp_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
14940   {
14941     \if_case:w #2 \exp_stop_f:
14942       \__fp_case_return_o:Nw \c_one_fp
14943     \or:
14944       \exp_after:wN \__fp_exp_normal:w
14945     \or:
14946       \if_meaning:w 0 #3
14947         \exp_after:wN \__fp_case_return_o:Nw
14948         \exp_after:wN \c_inf_fp
14949       \else:
14950         \exp_after:wN \__fp_case_return_o:Nw
14951         \exp_after:wN \c_zero_fp
14952       \fi:
14953     \or:
14954       \__fp_case_return_same_o:w
14955     \fi:
14956     \s__fp \__fp_chk:w #2#3#4;
14957   }
```

(End definition for __fp_exp_o:w*.)*

__fp_exp_normal:w
__fp_exp_pos:Nnwnw

```
14958 \cs_new:Npn \__fp_exp_normal:w \s__fp \__fp_chk:w 1#1
14959   {
14960     \if_meaning:w 0 #1
14961       \__fp_exp_pos:NNwnw + \__fp_fixed_to_float:wN
14962     \else:
14963       \__fp_exp_pos:NNwnw - \__fp_fixed_inv_to_float:wN
14964     \fi:
14965   }
14966 \cs_new:Npn \__fp_exp_pos:NNwnw #1#2#3 \fi: #4#5;
14967   {
14968     \fi:
14969     \exp_after:wN \__fp_sanitize:Nw
14970     \exp_after:wN 0
14971     \__int_value:w #1 \__int_eval:w
14972       \if_int_compare:w #4 < - \c_eight
14973         \c_one
14974         \exp_after:wN \__fp_add_big_i_o:wNww
14975         \__int_value:w \__int_eval:w \c_one - #4 ;
14976         0 {1000}{0000}{0000}{0000} ; #5;
14977         \exp:w
14978       \else:
14979         \if_int_compare:w #4 > \c_five % cf \c__fp_max_exponent_int
14980           \exp_after:wN \__fp_exp_overflow:
14981           \exp:w
14982         \else:
14983           \if_int_compare:w #4 < \c_zero
14984             \exp_after:wN \use_i:nn
```

```
14985          \else:
14986            \exp_after:wN \use_ii:nn
14987          \fi:
14988          {
14989            \c_zero
14990            \__fp_decimate:nNnnnn { - #4 }
14991              \__fp_exp_Taylor:Nnnwn
14992          }
14993          {
14994            \__fp_decimate:nNnnnn { \c_sixteen - #4 }
14995              \__fp_exp_pos_large:NnnNwn
14996          }
14997          #5
14998          {#4}
14999          #1 #2 0
15000          \exp:w
15001        \fi:
15002      \fi:
15003    \exp_after:wN \c_zero
15004  }
15005 \cs_new:Npn \__fp_exp_overflow:
15006  { + \c_two * \c__fp_max_exponent_int ; {1000} {0000} {0000} {0000} ; }
```

(*End definition for* `\__fp_exp_normal:w` *and* `\__fp_exp_pos:Nnnwn`.)

`\__fp_exp_Taylor:Nnnwn`
`\__fp_exp_Taylor_loop:www`
`\__fp_exp_Taylor_break:Nww`

This function is called for numbers in the range $[10^{-9}, 10^{-1})$. Our only task is to compute 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.

```
15007 \cs_new:Npn \__fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
15008  {
15009    #6
15010    \__fp_pack_twice_four:wNNNNNNNN
15011    \__fp_pack_twice_four:wNNNNNNNN
15012    \__fp_pack_twice_four:wNNNNNNNN
15013    \__fp_exp_Taylor_ii:ww
15014    ; #2#3#4 0000 0000 ;
15015  }
15016 \cs_new:Npn \__fp_exp_Taylor_ii:ww #1; #2;
15017  { \__fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s__stop }
15018 \cs_new:Npn \__fp_exp_Taylor_loop:www #1; #2; #3;
15019  {
15020    \if_int_compare:w #1 = \c_one
15021      \exp_after:wN \__fp_exp_Taylor_break:Nww
15022    \fi:
15023    \__fp_fixed_div_int:wwN #3 ; #1 ;
15024    \__fp_fixed_add_one:wN
15025    \__fp_fixed_mul:wwn #2 ;
15026    {
15027      \exp_after:wN \__fp_exp_Taylor_loop:www
15028      \__int_value:w \__int_eval:w #1 - 1 ;
15029      #2 ;
15030    }
15031  }
```

675

```
15032 \cs_new:Npn \__fp_exp_Taylor_break:Nww #1 #2; #3 \s__stop
15033     { \__fp_fixed_add_one:wN #2 ; }
```

(*End definition for* __fp_exp_Taylor:Nnnwn, __fp_exp_Taylor_loop:www, *and* __fp_exp_Taylor_-
break:Nww.)

<div style="float:left">

__fp_exp_pos_large:NnnNwn
__fp_exp_large_after:wwn
__fp_exp_large:w
__fp_exp_large_v:wN
__fp_exp_large_iv:wN
__fp_exp_large_iii:wN
__fp_exp_large_ii:wN
__fp_exp_large_i:wN
__fp_exp_large_:wN

</div>

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 $\exp(\langle digit \rangle \cdot 10^{\langle exponent \rangle})$ in a table, and multiplying that to the current total. The loop is done by having the auxiliary for one exponent call the auxiliary for the next exponent. The current total is expressed by leaving the exponent behind in the input stream (we are currently within an __int_-
eval:w), and keeping track of a fixed point number, #1 for the numbered auxiliaries. Our usage of \if_case:w is somewhat dirty for optimization: TeX jumps to the appropriate case, but we then close the \if_case:w "by hand", using \or: and \fi: as delimiters.

```
15034 \cs_new:Npn \__fp_exp_pos_large:NnnNwn #1#2#3 #4#5; #6
15035     {
15036         \exp_after:wN \exp_after:wN
15037         \cs:w __fp_exp_large_ \__int_to_roman:w #6 :wN \exp_after:wN \cs_end:
15038         \exp_after:wN \c__fp_one_fixed_tl
15039         \exp_after:wN ;
15040         \__int_value:w #3 #4 \exp_stop_f:
15041         #5 00000 ;
15042     }
15043 \cs_new:Npn \__fp_exp_large:w #1 \or: #2 \fi:
15044     { \fi: \__fp_fixed_mul:wwn #1; }
15045 \cs_new:Npn \__fp_exp_large_v:wN #1; #2
15046     {
15047         \if_case:w #2 ~             \exp_after:wN \__fp_fixed_continue:wn  \or:
15048             + 4343 \__fp_exp_large:w {8806}{8182}{2566}{2921}{5872}{6150} \or:
15049             + 8686 \__fp_exp_large:w {7756}{0047}{2598}{6861}{0458}{3204} \or:
15050             + 13029 \__fp_exp_large:w {6830}{5723}{7791}{4884}{1932}{7351} \or:
15051             + 17372 \__fp_exp_large:w {6015}{5609}{3095}{3052}{3494}{7574} \or:
15052             + 21715 \__fp_exp_large:w {5297}{7951}{6443}{0315}{3251}{3576} \or:
15053             + 26058 \__fp_exp_large:w {4665}{6719}{0099}{3379}{5527}{2929} \or:
15054             + 30401 \__fp_exp_large:w {4108}{9724}{3326}{3186}{5271}{5665} \or:
15055             + 34744 \__fp_exp_large:w {3618}{6973}{3140}{0875}{3856}{4102} \or:
15056             + 39087 \__fp_exp_large:w {3186}{9209}{6113}{3900}{6705}{9685} \or:
15057         \fi:
15058         #1;
15059         \__fp_exp_large_iv:wN
15060     }
15061 \cs_new:Npn \__fp_exp_large_iv:wN #1; #2
15062     {
15063         \if_case:w #2 ~             \exp_after:wN \__fp_fixed_continue:wn  \or:
15064             + 435 \__fp_exp_large:w {1970}{0711}{1401}{7046}{9938}{8888} \or:
15065             + 869 \__fp_exp_large:w {3881}{1801}{9428}{4368}{5764}{8232} \or:
15066             + 1303 \__fp_exp_large:w {7646}{2009}{8905}{4704}{8893}{1073} \or:
15067             + 1738 \__fp_exp_large:w {1506}{3559}{7005}{0524}{9009}{7592} \or:
15068             + 2172 \__fp_exp_large:w {2967}{6283}{8402}{3667}{0689}{6630} \or:
15069             + 2606 \__fp_exp_large:w {5846}{4389}{5650}{2114}{7278}{5046} \or:
15070             + 3041 \__fp_exp_large:w {1151}{7900}{5080}{6878}{2914}{4154} \or:
```

676

```
15071       + 3475 \__fp_exp_large:w {2269}{1083}{0850}{6857}{8724}{4002} \or:
15072       + 3909 \__fp_exp_large:w {4470}{3047}{3316}{5442}{6408}{6591} \or:
15073     \fi:
15074     #1;
15075     \__fp_exp_large_iii:wN
15076   }
15077 \cs_new:Npn \__fp_exp_large_iii:wN #1; #2
15078   {
15079     \if_case:w #2 ~          \exp_after:wN \__fp_fixed_continue:wn  \or:
15080       +  44 \__fp_exp_large:w {2688}{1171}{4181}{6135}{4484}{1263} \or:
15081       +  87 \__fp_exp_large:w {7225}{9737}{6812}{5749}{2581}{7748} \or:
15082       + 131 \__fp_exp_large:w {1942}{4263}{9524}{1255}{9365}{8421} \or:
15083       + 174 \__fp_exp_large:w {5221}{4696}{8976}{4143}{9505}{8876} \or:
15084       + 218 \__fp_exp_large:w {1403}{5922}{1785}{2837}{4107}{3977} \or:
15085       + 261 \__fp_exp_large:w {3773}{0203}{0092}{9939}{8234}{0143} \or:
15086       + 305 \__fp_exp_large:w {1014}{2320}{5473}{5004}{5094}{5533} \or:
15087       + 348 \__fp_exp_large:w {2726}{3745}{7211}{2566}{5673}{6478} \or:
15088       + 391 \__fp_exp_large:w {7328}{8142}{2230}{7421}{7051}{8866} \or:
15089     \fi:
15090     #1;
15091     \__fp_exp_large_ii:wN
15092   }
15093 \cs_new:Npn \__fp_exp_large_ii:wN #1; #2
15094   {
15095     \if_case:w #2 ~          \exp_after:wN \__fp_fixed_continue:wn  \or:
15096       +  5 \__fp_exp_large:w {2202}{6465}{7948}{0671}{6516}{9579} \or:
15097       +  9 \__fp_exp_large:w {4851}{6519}{5409}{7902}{7796}{9107} \or:
15098       + 14 \__fp_exp_large:w {1068}{6474}{5815}{2446}{2146}{9905} \or:
15099       + 18 \__fp_exp_large:w {2353}{8526}{6837}{0199}{8540}{7900} \or:
15100       + 22 \__fp_exp_large:w {5184}{7055}{2858}{7072}{4640}{8745} \or:
15101       + 27 \__fp_exp_large:w {1142}{0073}{8981}{5684}{2836}{6296} \or:
15102       + 31 \__fp_exp_large:w {2515}{4386}{7091}{9167}{0062}{6578} \or:
15103       + 35 \__fp_exp_large:w {5540}{6223}{8439}{3510}{0525}{7117} \or:
15104       + 40 \__fp_exp_large:w {1220}{4032}{9431}{7840}{8020}{0271} \or:
15105     \fi:
15106     #1;
15107     \__fp_exp_large_i:wN
15108   }
15109 \cs_new:Npn \__fp_exp_large_i:wN #1; #2
15110   {
15111     \if_case:w #2 ~          \exp_after:wN \__fp_fixed_continue:wn  \or:
15112       + 1 \__fp_exp_large:w {2718}{2818}{2845}{9045}{2353}{6029} \or:
15113       + 1 \__fp_exp_large:w {7389}{0560}{9893}{0650}{2272}{3043} \or:
15114       + 2 \__fp_exp_large:w {2008}{5536}{9231}{8766}{7740}{9285} \or:
15115       + 2 \__fp_exp_large:w {5459}{8150}{0331}{4423}{9078}{1103} \or:
15116       + 3 \__fp_exp_large:w {1484}{1315}{9102}{5766}{0342}{1116} \or:
15117       + 3 \__fp_exp_large:w {4034}{2879}{3492}{7351}{2260}{8387} \or:
15118       + 4 \__fp_exp_large:w {1096}{6331}{5842}{8458}{5992}{6372} \or:
15119       + 4 \__fp_exp_large:w {2980}{9579}{8704}{1728}{2747}{4359} \or:
15120       + 4 \__fp_exp_large:w {8103}{0839}{2757}{5384}{0077}{1000} \or:
15121     \fi:
15122     #1;
15123     \__fp_exp_large_:wN
15124   }
```

```
15125 \cs_new:Npn \__fp_exp_large_:wN #1; #2
15126   {
15127     \if_case:w #2 ~        \exp_after:wN \__fp_fixed_continue:wn  \or:
15128       + 1 \__fp_exp_large:w {1105}{1709}{1807}{5647}{6248}{1171} \or:
15129       + 1 \__fp_exp_large:w {1221}{4027}{5816}{0169}{8339}{2107} \or:
15130       + 1 \__fp_exp_large:w {1349}{8588}{0757}{6003}{1039}{8374} \or:
15131       + 1 \__fp_exp_large:w {1491}{8246}{9764}{1270}{3178}{2485} \or:
15132       + 1 \__fp_exp_large:w {1648}{7212}{7070}{0128}{1468}{4865} \or:
15133       + 1 \__fp_exp_large:w {1822}{1188}{0039}{0508}{9748}{7537} \or:
15134       + 1 \__fp_exp_large:w {2013}{7527}{0747}{0476}{5216}{2455} \or:
15135       + 1 \__fp_exp_large:w {2225}{5409}{2849}{2467}{6045}{7954} \or:
15136       + 1 \__fp_exp_large:w {2459}{6031}{1115}{6949}{6638}{0013} \or:
15137     \fi:
15138     #1;
15139     \__fp_exp_large_after:wwn
15140   }
15141 \cs_new:Npn \__fp_exp_large_after:wwn #1; #2; #3
15142   {
15143     \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2; {} #3
15144     \__fp_fixed_mul:wwn #1;
15145   }
```

(*End definition for* `\__fp_exp_pos_large:NnnNwn` *and others.*)

29.3   Power

Raising a number a to a power b leads to many distinct situations.

a^b	$-\infty$	$-y$	$-n$	± 0	$+n$	$+y$	$+\infty$	NaN
$+\infty$	$+0$	$+0$	$+0$	$+1$	$+\infty$	$+\infty$	$+\infty$	NaN
$1 < x$	$+0$	$+x^{-y}$	$+x^{-n}$	$+1$	$+x^n$	$+x^y$	$+\infty$	NaN
$+1$	$+1$	$+1$	$+1$	$+1$	$+1$	$+1$	$+1$	$+1$
$0 < x < 1$	$+\infty$	$+x^{-y}$	$+x^{-n}$	$+1$	$+x^n$	$+x^y$	$+0$	NaN
$+0$	$+\infty$	$+\infty$	$+\infty$	$+1$	$+0$	$+0$	$+0$	NaN
-0	NaN	NaN	$\pm\infty$	$+1$	± 0	$+0$	$+0$	NaN
$-1 < -x < 0$	NaN	NaN	$\pm x^{-n}$	$+1$	$\pm x^n$	NaN	$+0$	NaN
-1	NaN	NaN	± 1	$+1$	± 1	NaN	NaN	NaN
$-x < -1$	$+0$	NaN	$\pm x^{-n}$	$+1$	$\pm x^n$	NaN	NaN	NaN
$-\infty$	$+0$	$+0$	± 0	$+1$	$\pm\infty$	NaN	NaN	NaN
NaN	NaN	NaN	NaN	$+1$	NaN	NaN	NaN	NaN

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 a 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: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 $\mathtt{a}^b$ ($\verb|\__fp_pow_normal:ww|$ 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.

```
15146 \cs_new:cpn { __fp_ \iow_char:N \^ _o:ww }
15147     \s__fp \__fp_chk:w #1#2#3; \s__fp \__fp_chk:w #4#5#6;
15148   {
15149     \if_meaning:w 0 #4
15150       \__fp_case_return_o:Nw \c_one_fp
15151     \fi:
15152     \if_case:w #2 \exp_stop_f:
15153       \exp_after:wN \use_i:nn
15154     \or:
15155       \__fp_case_return_o:Nw \c_nan_fp
15156     \else:
15157       \exp_after:wN \__fp_pow_neg:www
15158       \exp:w \exp_end_continue_f:w \exp_after:wN \use:nn
15159     \fi:
15160     {
15161       \if_meaning:w 1 #1
15162         \exp_after:wN \__fp_pow_normal:ww
15163       \else:
15164         \exp_after:wN \__fp_pow_zero_or_inf:ww
15165       \fi:
15166       \s__fp \__fp_chk:w #1#2#3;
15167     }
15168     { \s__fp \__fp_chk:w #4#5#6; \s__fp \__fp_chk:w #1#2#3; }
15169     \s__fp \__fp_chk:w #4#5#6;
15170   }
```

(*End definition for* $\verb|\__fp_^_o:ww|$.)

$\verb|\__fp_pow_zero_or_inf:ww|$ Raising -0 or $-\infty$ to $\mathtt{nan}$ yields $\mathtt{nan}$. 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 $a = \pm 0$ with $b < 0$ and we have a division by zero, or $a = \pm\infty$ and $b > 0$ and the result is also $+\infty$, but without any exception.

```
15171 \cs_new:Npn \__fp_pow_zero_or_inf:ww
15172     \s__fp \__fp_chk:w #1#2; \s__fp \__fp_chk:w #3#4
15173   {
15174     \if_meaning:w 1 #4
15175       \__fp_case_return_same_o:w
15176     \fi:
15177     \if_meaning:w #1 #4
15178       \__fp_case_return_o:Nw \c_zero_fp
15179     \fi:
15180     \if_meaning:w 0 #1
15181       \__fp_case_use:nw
15182         {
15183           \__fp_division_by_zero_o:NNww \c_inf_fp ^
15184             \s__fp \__fp_chk:w #1 #2 ;
15185         }
15186     \else:
```

```
15187            \__fp_case_return_o:Nw \c_inf_fp
15188          \fi:
15189          \s__fp \__fp_chk:w #3#4
15190        }
```

(*End definition for* `\__fp_pow_zero_or_inf:ww`.)

`\__fp_pow_normal: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:ww`.

2 Return $+\infty$ or $+0$ depending on the sign of b and whether the exponent of a is positive or not.

3 Return b.

```
15191 \cs_new:Npn \__fp_pow_normal:ww
15192      \s__fp \__fp_chk:w 1 #1#2#3; \s__fp \__fp_chk:w #4#5
15193   {
15194      \if_int_compare:w \__str_if_eq_x:nn { #2 #3 }
15195              { 1 {1000} {0000} {0000} {0000} } = \c_zero
15196        \if_int_compare:w #4 #1 = 32 \exp_stop_f:
15197          \exp_after:wN \__fp_case_return_ii_o:ww
15198        \fi:
15199        \__fp_case_return_o:Nww \c_one_fp
15200      \fi:
15201      \if_case:w #4 \exp_stop_f:
15202      \or:
15203        \exp_after:wN \__fp_pow_npos:Nww
15204        \exp_after:wN #5
15205      \or:
15206        \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
15207        \if_int_compare:w #2 > \c_zero
15208          \exp_after:wN \__fp_case_return_o:Nww
15209          \exp_after:wN \c_inf_fp
15210        \else:
15211          \exp_after:wN \__fp_case_return_o:Nww
15212          \exp_after:wN \c_zero_fp
15213        \fi:
15214      \or:
15215        \__fp_case_return_ii_o:ww
15216      \fi:
15217      \s__fp \__fp_chk:w 1 #1 {#2} #3 ;
15218      \s__fp \__fp_chk:w #4 #5
15219   }
```

(*End definition for* `\__fp_pow_normal:ww`.)

`\__fp_pow_npos: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

680

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.

```
15220 \cs_new:Npn \__fp_pow_npos:Nww #1 \s__fp \__fp_chk:w 1#2#3
15221   {
15222     \exp_after:wN \__fp_sanitize:Nw
15223     \exp_after:wN 0
15224     \__int_value:w
15225       \if:w #1 \if_int_compare:w #3 > \c_zero 0 \else: 2 \fi:
15226         \exp_after:wN \__fp_pow_npos_aux:NNnww
15227         \exp_after:wN +
15228         \exp_after:wN \__fp_fixed_to_float:wN
15229       \else:
15230         \exp_after:wN \__fp_pow_npos_aux:NNnww
15231         \exp_after:wN -
15232         \exp_after:wN \__fp_fixed_inv_to_float:wN
15233       \fi:
15234       {#3}
15235   }
```

(*End definition for* `\__fp_pow_npos: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 $-\ln(x)$.

```
15236 \cs_new:Npn \__fp_pow_npos_aux:NNnww #1#2#3#4#5; \s__fp \__fp_chk:w 1#6#7#8;
15237   {
15238     #1
15239     \__int_eval:w
15240       \__fp_ln_significand:NNNNnnnN #4#5
15241       \__fp_pow_exponent:wnN {#3}
15242       \__fp_fixed_mul:wwn #8 {0000}{0000} ;
15243       \__fp_pow_B:wwN #7;
15244       #1 #2 0 % fixed_to_float:wN
15245   }
15246 \cs_new:Npn \__fp_pow_exponent:wnN #1; #2
15247   {
15248     \if_int_compare:w #2 > \c_zero
15249       \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % n\ln(10) - (-\ln(x))
15250       \exp_after:wN +
15251     \else:
15252       \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % -(|n|\ln(10) + (-\ln(x)))
15253       \exp_after:wN -
15254     \fi:
15255     #2; #1;
15256   }
15257 \cs_new:Npn \__fp_pow_exponent:Nwnnnnnw #1#2; #3#4#5#6#7#8;
15258   { %^^A todo: use that in ln.
15259     \exp_after:wN \__fp_fixed_mul_after:wwn
15260     \__int_value:w \__int_eval:w \c__fp_leading_shift_int
15261       \exp_after:wN \__fp_pack:NNNNNw
15262       \__int_value:w \__int_eval:w \c__fp_middle_shift_int
15263         #1#2*23025 - #1 #3
```

```
15264        \exp_after:wN \__fp_pack:NNNNNw
15265        \__int_value:w \__int_eval:w \c__fp_middle_shift_int
15266          #1 #2*8509 - #1 #4
15267          \exp_after:wN \__fp_pack:NNNNNw
15268          \__int_value:w \__int_eval:w \c__fp_middle_shift_int
15269            #1 #2*2994 - #1 #5
15270            \exp_after:wN \__fp_pack:NNNNNw
15271            \__int_value:w \__int_eval:w \c__fp_middle_shift_int
15272              #1 #2*0456 - #1 #6
15273              \exp_after:wN \__fp_pack:NNNNNw
15274              \__int_value:w \__int_eval:w \c__fp_trailing_shift_int
15275                #1 #2*8401 - #1 #7
15276                #1 ( #2*7991 - #8 ) / 1 0000 ; ;
15277    }
15278 \cs_new:Npn \__fp_pow_B:wwN #1#2#3#4#5#6; #7;
15279    {
15280      \if_int_compare:w #7 < \c_zero
15281        \exp_after:wN \__fp_pow_C_neg:w \__int_value:w -
15282      \else:
15283        \if_int_compare:w #7 < 22 \exp_stop_f:
15284          \exp_after:wN \__fp_pow_C_pos:w \__int_value:w
15285        \else:
15286          \exp_after:wN \__fp_pow_C_overflow:w \__int_value:w
15287        \fi:
15288      \fi:
15289      #7 \exp_after:wN ;
15290      \__int_value:w \__int_eval:w 10 0000 + #1 \__int_eval_end:
15291      #2#3#4#5#6 0000 0000 0000 0000 0000 0000 ; %^^A todo: how many 0?
15292    }
15293 \cs_new:Npn \__fp_pow_C_overflow:w #1; #2; #3
15294    {
15295      + \c_two * \c__fp_max_exponent_int
15296      \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl ;
15297    }
15298 \cs_new:Npn \__fp_pow_C_neg:w #1 ; 1
15299    {
15300      \exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w
15301      \prg_replicate:nn {#1} {0}
15302    }
15303 \cs_new:Npn \__fp_pow_C_pos:w #1; 1
15304    { \__fp_pow_C_pos_loop:wN #1; }
15305 \cs_new:Npn \__fp_pow_C_pos_loop:wN #1; #2
15306    {
15307      \if_meaning:w 0 #1
15308        \exp_after:wN \__fp_pow_C_pack:w
15309        \exp_after:wN #2
15310      \else:
15311        \if_meaning:w 0 #2
15312          \exp_after:wN \__fp_pow_C_pos_loop:wN \__int_value:w
15313        \else:
15314          \exp_after:wN \__fp_pow_C_overflow:w \__int_value:w
15315        \fi:
15316        \__int_eval:w #1 - \c_one \exp_after:wN ;
15317      \fi:
```

682

```
15318     }
15319 \cs_new:Npn \__fp_pow_C_pack:w
15320     { \exp_after:wN \__fp_exp_large_v:wN \c__fp_one_fixed_tl ; }
```

(*End 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), $\mathtt{a}^b = \mathtt{a}^b$. If b is an odd integer (case 0), $\mathtt{a}^b = -\mathtt{a}^b$, 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 $\mathtt{nan}$, in which case we return that as $\mathtt{a}^b$. In particular, since the underflow detection occurs before __fp_pow_neg:www is called, (-0.1)**(12345.6) will give $+0$ rather than complaining that the sign is not defined.

```
15321 \cs_new:Npn \__fp_pow_neg:www \s__fp \__fp_chk:w #1#2; #3; #4;
15322     {
15323         \if_case:w \__fp_pow_neg_case:w #4 ;
15324             \exp_after:wN \__fp_pow_neg_aux:wNN
15325         \or:
15326             \if_int_compare:w \__int_eval:w #1 / \c_two = \c_one
15327                 \__fp_invalid_operation_o:Nww ^ #3; #4;
15328                 \exp:w \exp_end_continue_f:w
15329                 \exp_after:wN \exp_after:wN
15330                 \exp_after:wN \__fp_use_none_until_s:w
15331             \fi:
15332         \fi:
15333         \__fp_exp_after_o:w
15334         \s__fp \__fp_chk:w #1#2;
15335     }
15336 \cs_new:Npn \__fp_pow_neg_aux:wNN #1 \s__fp \__fp_chk:w #2#3
15337     {
15338         \exp_after:wN \__fp_exp_after_o:w
15339         \exp_after:wN \s__fp
15340         \exp_after:wN \__fp_chk:w
15341         \exp_after:wN #2
15342         \__int_value:w \__int_eval:w \c_two - #3 \__int_eval_end:
15343     }
```

(*End 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:NNNNNNNNw

This function expects a floating point number, and "returns" -1 if it is an even integer, 0 if it is an odd integer, and 1 if it is not an integer. Zeros are even, $\pm\infty$ and $\mathtt{nan}$ are non-integers. The sign of normal numbers is irrelevant to parity. If the exponent is greater than sixteen, then the number is even. If the exponent is non-positive, the number cannot be an integer. We also separate the ranges of exponent $[1, 8]$ and $[9, 16]$. In the former case, check that the last 8 digits are zero (otherwise we don't have an integer). In both cases, consider the appropriate 8 digits, either #4#5 or #2#3, remove the first few: we are then left with $\langle digit \rangle \langle digits \rangle$; which would be the digits surrounding the decimal period. If the $\langle digits \rangle$ are non-zero, the number is not an integer. Otherwise, check the parity of the $\langle digit \rangle$ and return \c_zero or \c_minus_one.

```
15344 \cs_new:Npn \__fp_pow_neg_case:w \s__fp \__fp_chk:w #1#2#3;
15345     {
15346         \if_case:w #1 \exp_stop_f:
15347                 \c_minus_one
15348         \or:    \__fp_pow_neg_case_aux:nnnnn #3
```

683

```
15349          \else: \c_one
15350        \fi:
15351      }
15352    \cs_new:Npn \__fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
15353      {
15354        \if_int_compare:w #1 > \c_eight
15355          \if_int_compare:w #1 > \c_sixteen
15356            \c_minus_one
15357          \else:
15358            \exp_after:wN \exp_after:wN
15359            \exp_after:wN \__fp_pow_neg_case_aux:NNNNNNNNw
15360            \prg_replicate:nn { \c_sixteen - #1 } { 0 } #4#5 ;
15361          \fi:
15362        \else:
15363          \if_int_compare:w #1 > \c_zero
15364            \if_int_compare:w #4#5 = \c_zero
15365              \exp_after:wN \exp_after:wN
15366              \exp_after:wN \__fp_pow_neg_case_aux:NNNNNNNNw
15367              \prg_replicate:nn { \c_eight - #1 } { 0 } #2#3 ;
15368            \else:
15369              \c_one
15370            \fi:
15371          \else:
15372            \c_one
15373          \fi:
15374        \fi:
15375      }
15376    \cs_new:Npn \__fp_pow_neg_case_aux:NNNNNNNNw #1#2#3#4#5#6#7#8#9;
15377      {
15378        \if_int_compare:w 0 #9 = \c_zero
15379          \if_int_odd:w #8 \exp_stop_f:
15380            \c_zero
15381          \else:
15382            \c_minus_one
15383          \fi:
15384        \else:
15385          \c_one
15386        \fi:
15387      }
```

(*End definition for* __fp_pow_neg_case:w, __fp_pow_neg_case_aux:nnnnn, *and* __fp_pow_neg_-case_aux:NNNNNNNNw.)

```
15388  ⟨/initex | package⟩
```

30   l3fp-trig Implementation

```
15389  ⟨*initex | package⟩
```

```
15390  ⟨@@=fp⟩
```

30.1   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 $\lfloor \frac{x}{\pi/4} \rfloor \mod 8$ (in degrees, the same formula with $\pi/4 \to 45$), the sign, and the function to compute.

30.1.1 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` will be called to compute the Taylor series: this function receives a sign `#3`, an initial octant of 0, and the function `\__fp_ep_to_float:wwN` which converts the result of the series to a floating point directly rather than taking its inverse, since $\sin(x) = \#3 \sin|x|$.

```
15391 \cs_new:Npn \__fp_sin_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
15392   {
15393     \if_case:w #2 \exp_stop_f:
15394           \__fp_case_return_same_o:w
15395     \or:   \__fp_case_use:nw
15396            {
15397              \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
15398              \__fp_ep_to_float:wwN #3 \c_zero
15399            }
15400     \or:   \__fp_case_use:nw
15401            { \__fp_invalid_operation_o:fw { #1 { sin } { sind } } }
15402     \else: \__fp_case_return_same_o:w
15403     \fi:
15404     \s__fp \__fp_chk:w #2 #3 #4;
15405   }
```

(End 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 will 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 $\cos(x) = +\sin(\pi/2 + |x|)$.

```
15406 \cs_new:Npn \__fp_cos_o:w #1 \s__fp \__fp_chk:w #2#3; @
15407   {
15408     \if_case:w #2 \exp_stop_f:
```

```
15409                \__fp_case_return_o:Nw \c_one_fp
15410       \or:   \__fp_case_use:nw
15411                {
15412                    \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
15413                    \__fp_ep_to_float:wwN 0 \c_two
15414                }
15415       \or:   \__fp_case_use:nw
15416                { \__fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
15417       \else: \__fp_case_return_same_o:w
15418       \fi:
15419       \s__fp \__fp_chk:w #2 #3;
15420    }
```

(*End 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 $\csc(x) = \#3\bigl(\sin|x|\bigr)^{-1}$.

```
15421 \cs_new:Npn \__fp_csc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
15422    {
15423       \if_case:w #2 \exp_stop_f:
15424                \__fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
15425       \or:   \__fp_case_use:nw
15426                {
15427                    \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
15428                    \__fp_ep_inv_to_float:wwN #3 \c_zero
15429                }
15430       \or:   \__fp_case_use:nw
15431                { \__fp_invalid_operation_o:fw { #1 { csc } { cscd } } }
15432       \else: \__fp_case_return_same_o:w
15433       \fi:
15434       \s__fp \__fp_chk:w #2 #3 #4;
15435    }
```

(*End 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 $\sec(x) = +1/\sin(\pi/2 + |x|)$.

```
15436 \cs_new:Npn \__fp_sec_o:w #1 \s__fp \__fp_chk:w #2#3; @
15437    {
15438       \if_case:w #2 \exp_stop_f:
15439                \__fp_case_return_o:Nw \c_one_fp
15440       \or:   \__fp_case_use:nw
15441                {
15442                    \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
15443                    \__fp_ep_inv_to_float:wwN 0 \c_two
15444                }
15445       \or:   \__fp_case_use:nw
```

```
15446                    { \__fp_invalid_operation_o:fw { #1 { sec } { secd } } } }
15447        \else: \__fp_case_return_same_o:w
15448        \fi:
15449        \s__fp \__fp_chk:w #2 #3;
15450      }
```

(*End 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.

```
15451  \cs_new:Npn \__fp_tan_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
15452      {
15453        \if_case:w #2 \exp_stop_f:
15454                    \__fp_case_return_same_o:w
15455        \or:    \__fp_case_use:nw
15456                  {
15457                    \__fp_trig:NNNNNwn #1
15458                      \__fp_tan_series_o:NNwwww 0 #3 \c_one
15459                  }
15460        \or:    \__fp_case_use:nw
15461                  { \__fp_invalid_operation_o:fw { #1 { tan } { tand } } } }
15462        \else: \__fp_case_return_same_o:w
15463        \fi:
15464        \s__fp \__fp_chk:w #2 #3 #4;
15465      }
```

(*End 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.

```
15466  \cs_new:Npn \__fp_cot_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
15467      {
15468        \if_case:w #2 \exp_stop_f:
15469                    \__fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
15470        \or:    \__fp_case_use:nw
15471                  {
15472                    \__fp_trig:NNNNNwn #1
15473                      \__fp_tan_series_o:NNwwww 2 #3 \c_three
15474                  }
15475        \or:    \__fp_case_use:nw
15476                  { \__fp_invalid_operation_o:fw { #1 { cot } { cotd } } } }
15477        \else: \__fp_case_return_same_o:w
15478        \fi:
15479        \s__fp \__fp_chk:w #2 #3 #4;
15480      }
15481  \cs_new:Npn \__fp_cot_zero_o:Nfw #1#2#3 \fi:
```

```
15482    {
15483      \fi:
15484      \token_if_eq_meaning:NNTF 0 #1
15485        { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_inf_fp }
15486        { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_minus_inf_fp }
15487      {#2}
15488    }
```

(*End definition for* `\__fp_cot_o:w` *and* `\__fp_cot_zero_o:Nfw`.)

30.1.2 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:wN` or `\__fp_ep_inv_-to_float: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.

```
15489  \cs_new:Npn \__fp_trig:NNNNNwn #1#2#3#4#5 \s__fp \__fp_chk:w 1#6#7#8;
15490    {
15491      \exp_after:wN #2
15492      \exp_after:wN #3
15493      \exp_after:wN #4
15494      \__int_value:w \__int_eval:w #5
15495        \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
15496        \if_int_compare:w #7 > #1 \c_zero \c_one
15497          #1 \__fp_trig_large:ww \__fp_trigd_large:ww
15498        \else:
15499          #1 \__fp_trig_small:ww \__fp_trigd_small:ww
15500        \fi:
15501      #7,#8{0000}{0000};
15502    }
```

(*End definition for* `\__fp_trig:NNNNNwn`.)

30.1.3 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 will be 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.

```
15503 \cs_new:Npn \__fp_trig_small:ww #1,#2;
15504   { \__fp_ep_to_fixed:wwn #1,#2; . #1,#2; }
```

(*End 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.

```
15505 \cs_new:Npn \__fp_trigd_small:ww #1,#2;
15506   {
15507     \__fp_ep_mul_raw:wwwwN
15508       -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;
15509     \__fp_trig_small:ww
15510   }
```

(*End definition for* `\__fp_trigd_small:ww`.)

30.1.4 Argument reduction in degrees

`\__fp_trigd_large:ww`
`\_fp_trigd_large_auxi:nnnwNNNN`
`\__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 $100 \times \langle block_1 \rangle \cdots \langle block_3 \rangle . \langle block_4 \rangle \cdots \langle block_7 \rangle$, or is equal to it modulo 360 if the exponent `#1` is very large. The first auxiliary finds $\langle block_1 \rangle + \langle block_2 \rangle$ (mod 9), a single digit, and prepends it to the 4 digits of $\langle block_3 \rangle$. It also unpacks $\langle block_4 \rangle$ and grabs the 4 digits of $\langle block_7 \rangle$. The second auxiliary grabs the $\langle block_3 \rangle$ plus any contribution from the first two blocks as `#1`, the first digit of $\langle block_4 \rangle$ (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 $\langle block_4 \rangle$. 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 $\langle blocks \rangle$. 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).

```
15511 \cs_new:Npn \__fp_trigd_large:ww #1, #2#3#4#5#6#7;
15512   {
15513     \exp_after:wN \__fp_pack_eight:wNNNNNNNN
15514     \exp_after:wN \__fp_pack_eight:wNNNNNNNN
15515     \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
15516     \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
15517     \exp_after:wN \__fp_trigd_large_auxi:nnnwNNNN
15518     \exp_after:wN ;
15519     \exp:w \exp_end_continue_f:w
15520     \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
15521     #2#3#4#5#6#7 0000 0000 0000 !
15522   }
15523 \cs_new:Npn \__fp_trigd_large_auxi:nnnwNNNN #1#2#3#4#5; #6#7#8#9
15524   {
15525     \exp_after:wN \__fp_trigd_large_auxii:wNw
15526     \__int_value:w \__int_eval:w #1 + #2
15527       - (#1 + #2 - \c_four) / \c_nine * \c_nine \__int_eval_end:
15528     #3;
```

689

```
15529        #4; #5{#6#7#8#9};
15530      }
15531   \cs_new:Npn \__fp_trigd_large_auxii:wNw #1; #2#3;
15532      {
15533        + (#1#2 - \c_four) / \c_nine * \c_two
15534        \exp_after:wN \__fp_trigd_large_auxiii:www
15535          \__int_value:w \__int_eval:w #1#2
15536            - (#1#2 - \c_four) / \c_nine * \c_nine \__int_eval_end: #3 ;
15537      }
15538   \cs_new:Npn \__fp_trigd_large_auxiii:www #1; #2; #3!
15539      {
15540        \if_int_compare:w #1 < 4500 \exp_stop_f:
15541          \exp_after:wN \__fp_use_i_until_s:nw
15542          \exp_after:wN \__fp_fixed_continue:wn
15543        \else:
15544          + \c_one
15545        \fi:
15546        \__fp_fixed_sub:wwn {9000}{0000}{0000}{0000}{0000}{0000};
15547          {#1}#2{0000}{0000};
15548        { \__fp_trigd_small:ww 2, }
15549      }
```

(*End definition for* __fp_trigd_large:ww *and others.*)

30.1.5 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 $0.6 \cdot \text{ulp}$ in all cases.

__fp_trig_inverse_two_pi: This macro expands to ,,! or ,! followed by 10112 decimals of $10^{-16}/(2\pi)$. The number of decimals we really need is the maximum exponent plus the number of digits we will need later, 52, plus 12 ($4 - 1$ groups of 4 digits). We store the decimals as a control sequence name, and convert it to a token list when required: strings take up less memory than their token list representation.

```
15550   \cs_new:Npx \__fp_trig_inverse_two_pi:
15551      {
```

```
15552    \exp_not:n { \exp_after:wN \use_none:n \token_to_str:N }
15553    \cs:w , , !
15554    00000000000000001591549430918953357688837633725143620344 59645740 ~
15555    45644874766734405889679763422653509011380276625308595607 28427267 ~
15556    57958036892911846114578652877967410731699839229239966937 40907757 ~
15557    30777463969253076887173928962173976616933623902417236290 11832380 ~
15558    11422269975571594046189008690267395612048941093693784408 55287230 ~
15559    99946443400248672347739459610898323096783074906166986462 80469944 ~
15560    86521878815747865669642410389958741393486099838680991999 62442875 ~
15561    58517117885843111751876716054654753698800973946036475933 37680593 ~
15562    02494496635305327156775503220324777816397166022946748119 59816584 ~
15563    06060168030359981339119874988327866544352797550700162406 77564388 ~
15564    84957131088012219937614768137776473789063306804645797848 17613124 ~
15565    27314069960775024500297759857089056902796785131525210016 31774602 ~
15566    09248116062405614562031464840892484591914352115754075562 00871526 ~
15567    60680221715914075747458272259774628539987515532939081398 17724093 ~
15568    58254797073328719040699975907657707849347039358982808717 34256403 ~
15569    66895116625457059433276312686500261227179711532112599504 38667945 ~
15570    03762556083631711695259758128224941623334314510612353687 85631136 ~
15571    36692167142069746960129250578336053119608594509839556718 70995474 ~
15572    65104316238155175808394429799709950525438756612944588330 68460507 ~
15573    78529151514104048929885063881607761969930734103899957869 18905980 ~
15574    93737772061875432227189301366255261238780387538881106814 06765434 ~
15575    08282785269334267995560707903860603527389962451259957492 76297023 ~
15576    59409558430116482964118557771240575444945702178976979240 94903272 ~
15577    94770216649603565318153544003840689874717691588763190966 50696440 ~
15578    47769706876836567781047797954503533957583018818386879377 66124814 ~
15579    95305996558021908359875103512712904323158049871968687775 94656634 ~
15580    62210342044408554978503792738694293536619377829287359378 43470323 ~
15581    02371458379235571186363419294601831822919641650087830793 31353497 ~
15582    79099745864929026745060989368904588305033730305380547312 32158094 ~
15583    31976760322831314189809749822438335174356989847501039500 68388003 ~
15584    97867235996080240027390108749548547879235682611399489032 68997427 ~
15585    08349611492082890377678474303550456845608367147930845672 33270354 ~
15586    85392556202086839324099562211753318394020970793570774965 49880868 ~
15587    60663609686619670374745421028312192518462248349911611495 66556037 ~
15588    96967613993128299607760827799010078303600233827298790854 02387615 ~
15589    57445430926011910054337998389046549212482951607072853005 22721023 ~
15590    60175233131731797593110503281551093739136396453057926071 80083617 ~
15591    95487672464598047397729244810920093712578691833289588628 39904358 ~
15592    68666639756734451409503637327191743113880663830725923027 59734506 ~
15593    05482127780370653377830321709877349665684908003269885067 41791464 ~
15594    68350828161685331433616073099514985311981973375844420984 16559541 ~
15595    52250643394312864440383883561508797716450170647067518774 56059160 ~
15596    87168578579392262347563317111329986559415968907198506887 44230057 ~
15597    51919770569003821839256220338742353625680835415651729710 88117217 ~
15598    95936832564885187499748708553116598306101392144544601614 88452770 ~
15599    25114110702485217397451038667364038728600996748931735618 12071174 ~
15600    04788993688865569230784850230570571440636386320236852010 74100574 ~
15601    85922811157219680039782475953001669585221230346418773650 43546764 ~
15602    64565659719011230847670993097085912836466691917769387914 33315566 ~
15603    50669813216415210089571172862384260706784517601113450800 69947684 ~
15604    22356989624880515775980953397080854750597536265649034394 45420581 ~
15605    78864356830420003150955947434392525448506749142908647514 42303321 ~
```

```
15606    3324569511634945677539394240360905438335528292434220349484366151  ~
15607    4663228602477666660495314065734357553014090827988091478669343492  ~
15608    2737602634997829957018161964321233140475762897484082891174097478  ~
15609    2637899181699939487497715198981872666294601830539583275209236350  ~
15610    6853889228468247259972528300766856937583659722918244297474 06163  ~
15611    8183113958306744348516928597383237392662402434501997809940402189  ~
15612    6134834273613676449913827154166063424829363741850612261086132119  ~
15613    9863346284709941839942742955915628333990480382117501161211667205  ~
15614    1912579303552929241134403116134112495318385926958490443846807849  ~
15615    0973982808855297045153053991400988698840883654836652224668624087  ~
15616    2540140400911787421220452307533473972538149403884190586842311594  ~
15617    6322744339066125162393106283195323883392131534556381511752035108  ~
15618    7459558201123754359768155340187407394340363397803881721004531691  ~
15619    8295194879591767395417787924352761740724605939160273228287946819  ~
15620    3649128949714953432552723591659298072479985806126900733218844526  ~
15621    7943350455801952492566306204876616134365339920287545208555344144  ~
15622    0990512982727454659118132223284051166615650709837557433729548631  ~
15623    2041121716380915606161165732000083306114606181280326258695951602  ~
15624    4632166138576614804719932707771316441201594960110632830520759583  ~
15625    4850305079095584982982186740289838551383239570208076397550429225  ~
15626    9847647071016426974384504309165864528360324933604354657237557916  ~
15627    1366324120457809967156634022158805457943132827800552461320 88901  ~
15628    8742121092448910410052154968097113720754005710963406643135745439  ~
15629    9159769435788920793425617783022237011486424925239248728713132021  ~
15630    7667360756645598272609574156602343787436291321097485897150713073  ~
15631    9104072643541417970572226547980381512759579124002534468048220261  ~
15632    7342299001020483062463033796474678190501811830375153802879523433  ~
15633    4195502135689770912905614317878792086205744999257897569018492103  ~
15634    2420647138519113881475640209760554895793785141404145305151583964  ~
15635    2823265460206033118915865702720862502699163937515278873606 08114  ~
15636    5569484210322407727274216513642343669927163403094053074806 52685  ~
15637    0930165892136921414312937134106157153714062039784761842650297807  ~
15638    8606266969960809184223476335047746719017450451446166382846208240  ~
15639    8673595102371302904443779408535034454426334130626307459513830310  ~
15640    2293146934466832851766328241515210179422644395718121717021756492  ~
15641    1964449396532222187658488244511909401340504432139858628621083179  ~
15642    3939608443898019147873897723310286310131486955212620518278063494  ~
15643    5711866277825659883100535155231665984394090221806314454521212978  ~
15644    9734471488741258268223860236027109981191520568823472398358013366  ~
15645    0683786328867928619732367253606685216856320119489780733958419190  ~
15646    6659583867852941241871821727987506103946064819585745620060892122  ~
15647    8416394373846549589932028481236433466119707324309545859073361878  ~
15648    6290631850165106267576851216357588696307451999220010776676830946  ~
15649    9814975622682434793671310841210219520899481912444048751171059184  ~
15650    4139907889455775184621619041530934543802808938628073237578615267  ~
15651    7971143323241969857805637630180884386640607175368321362629671224  ~
15652    2609428540110963218262765120117022552929289655594608204938409069  ~
15653    0760692003954646191640021567336017909631872891998634341086903200  ~
15654    5796637103128612356988817640364252540837098108148351903121318624  ~
15655    7228181050845123690190646632235938872454630737272808789830041018  ~
15656    9485913673742589418124056729191238003306344998219631580386381054  ~
15657    2457893450084553280313511884341007373060595654437362488771292628  ~
15658    9807423539074061786905784443105274262641767830058221486462289361  ~
15659    9296692992033046693328438158053564864073184440599549689353773183  ~
```

692

```
67266131301086235880212880432893445621404797894542337360585063 27 ~
04399819326359166873419436567839012819122028162295003330122360 91 ~
85875592019590812241536794990954488810997589198908115811635388 91 ~
63394029237220498483752242362091008340975667917100841679570223 31 ~
78971071029288848970130995339954244153350606258439214524338646 40 ~
34324406573174775534054044810061776125690847464614329765439000 08 ~
38265211452101623664311197987319027511914412136169620456936026 33 ~
61023559621404670290121567964187357468358731723310047459633397 73 ~
24770449188851344153637600915375642674384501662213937193067487 06 ~
28815954648197751922077102367432890626907091179194127762122451 17 ~
23546771156404333577206166615646744746273056229133320309533405 51 ~
38417181946053215014263280008795518132967549728467018836574253 42 ~
50169942310691563431066260434122052138315879711507545406329065 7 ~
02484886486974028720372598692811493606274038423328749423321785 78 ~
77507355718570437873796934023369029114496144864976971943452746 7 ~
44296030894371925405266588907106620625755099303799766583679361 12 ~
81374511049715061537837435795558679721293587644630937572032213 20 ~
24605656611299713102758691128460432518434326915529284585734959 71 ~
50425565399302112184947232132380516549802909919676815118022483 192 ~
51273721997921343310676421874844262159851216763967793529829851 95 ~
85453921069578805868531232775454332291619890531890537253915822 22 ~
92325972781334278182560648823337607196810144814531983362379107 67 ~
12550175288263518364921035725874103565738946948754446940181759 23 ~
06093708281465018574253249692127646242478322107654737505681988 34 ~
56410354580272612522855031543250395918489189826304987591154063 21 ~
03542638900128374261551878773183758623551753785069565995700280 11 ~
58412588701500301702591674630208424124491283923380525772514737 141 ~
23102301725639683055535832628403836381576868284643304568059940 18 ~
70010719520929701779905832164175798681165865471477489647165479 48 ~
83121404318360798443140557311793496777637398989302277656070585 30 ~
40837477526409474350703952145247016838840709087061471944372256 50 ~
28231458729958697383168971268519390422971107213507569780372625 45 ~
81410950382703889873645162848201804682882058291353390138356491 44 ~
30040157065098879267154174507066868887834380555835011967458623 40 ~
80595327247278438292593957715840368859409899392552416883787935 72 ~
79679516540766739270312564187609621902430469934858919906001297 7 ~
74692145329704216778172615178506530085525599979402099694554315 45 ~
27458567044036668042864840451288118230979349696272183649293551 6 ~
20298724695832994819329783358034590232270526125421144370843595 84 ~
94433836383883177518411608817112512792333745772193398201900540 6 ~
32929377753069066074153049976826471244077688172486734216858815 09 ~
91334220759309471738551593408089571244106347208931949128807835 76 ~
31158294005497089180233665960770709275990105270281508688978285 49 ~
43403726427292621034870139928688535500620615143430786653960859 95 ~
00587149391416520653020700852656240747036607366053338052637667 57 ~
20188394972770472221536338511354834636246198554259938719333674 82 ~
04220974499566727025054464232439575068695913301937469191429809 99 ~
34242305501726652120924145596259605544275909519968243130842796 93 ~
71132070210498232381957471759855195018646309402975943631944500 91 ~
91506160492287643231921297034460935842592672763868143633098568 53 ~
27860243321410523307606588414958587181970712429959592267811727 96 ~
44388537967631392743142279531145000649221265001332686230215508 37
\cs_end:
}
```

(*End definition for* `\__fp_trig_inverse_two_pi:`.)

`\__fp_trig_large:ww`
`\__fp_trig_large_auxi:wwwww`
`\__fp_trig_large_auxii:ww`
`\__fp_trig_large_auxiii:wNNNNNNNN`
`\__fp_trig_large_auxiv:wN`

The exponent #1 is between 1 and 10000. We discard the integer part of $10^{\#1-16}/(2\pi)$, that is, the first #1 digits of $10^{-16}/(2\pi)$, because it yields an integer contribution to $x/(2\pi)$. The `auxii` auxiliary discards 64 digits at a time thanks to spaces inserted in the result of `\__fp_trig_inverse_two_pi:`, while `auxiii` discards 8 digits at a time, and `auxiv` discards digits one at a time. Then 64 digits are packed into groups of 4 and the `auxv` auxiliary is called.

```
15714 \cs_new:Npn \__fp_trig_large:ww #1, #2#3#4#5#6;
15715   {
15716     \exp_after:wN \__fp_trig_large_auxi:wwwww
15717     \__int_value:w \__int_eval:w (#1 - 32) / 64 \exp_after:wN ,
15718     \__int_value:w \__int_eval:w (#1 - 4) / 8 \exp_after:wN ,
15719     \__int_value:w #1 \__fp_trig_inverse_two_pi: ;
15720     {#2}{#3}{#4}{#5} ;
15721   }
15722 \cs_new:Npn \__fp_trig_large_auxi:wwwww #1, #2, #3, #4!
15723   {
15724     \prg_replicate:nn {#1} { \__fp_trig_large_auxii:ww }
15725     \prg_replicate:nn { #2 - #1 * \c_eight }
15726       { \__fp_trig_large_auxiii:wNNNNNNNN }
15727     \prg_replicate:nn { #3 - #2 * \c_eight }
15728       { \__fp_trig_large_auxiv:wN }
15729     \prg_replicate:nn { \c_eight } { \__fp_pack_twice_four:wNNNNNNNN }
15730     \__fp_trig_large_auxv:www
15731     ;
15732   }
15733 \cs_new:Npn \__fp_trig_large_auxii:ww #1; #2 ~ { #1; }
15734 \cs_new:Npn \__fp_trig_large_auxiii:wNNNNNNNN
15735     #1; #2#3#4#5#6#7#8#9 { #1; }
15736 \cs_new:Npn \__fp_trig_large_auxiv:wN #1; #2 { #1; }
```

(*End 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 some 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.

```
15737 \cs_new:Npn \__fp_trig_large_auxv:www #1; #2; #3;
15738   {
15739     \exp_after:wN \__fp_use_i_until_s:nw
15740     \exp_after:wN \__fp_trig_large_auxvii:w
15741     \__int_value:w \__int_eval:w \c__fp_leading_shift_int
15742       \prg_replicate:nn { \c_thirteen }
15743         { \__fp_trig_large_auxvi:wnnnnnnnn }
```

694

```
15744         + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
15745         \__fp_use_i_until_s:nw
15746       ; #3 #1 ; ;
15747     }
15748 \cs_new:Npn \__fp_trig_large_auxvi:wnnnnnnnn #1; #2#3#4#5#6#7#8#9
15749     {
15750       \exp_after:wN \__fp_trig_large_pack:NNNNNw
15751       \__int_value:w \__int_eval:w \c__fp_middle_shift_int
15752         + #2*#9 + #3*#8 + #4*#7 + #5*#6
15753       #1; {#2}{#3}{#4}{#5} {#7}{#8}{#9}
15754     }
15755 \cs_new:Npn \__fp_trig_large_pack:NNNNNw #1#2#3#4#5#6;
15756     { + #1#2#3#4#5 ; #6 }
```

(*End definition for* __fp_trig_large_auxv:www, __fp_trig_large_auxvi:wnnnnnnnn, *and* __fp_-
trig_large_pack:NNNNNw.)

__fp_trig_large_auxvii:w The auxvii auxiliary is followed by 52 digits and a semicolon. We find the octant as the
__fp_trig_large_auxviii:w integer part of 8 times what follows, or equivalently as the integer part of `#1#2#3/125`,
__fp_trig_large_auxix:Nw and add it to the surrounding integer expression for the octant. We then compute 8 times
__fp_trig_large_auxx:wNNNNN the 52-digit number, with a minus sign if the octant is odd. Again, the last `middle` shift
__fp_trig_large_auxxi:w 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.

```
15757 \cs_new:Npn \__fp_trig_large_auxvii:w #1#2#3
15758     {
15759       \exp_after:wN \__fp_trig_large_auxviii:ww
15760       \__int_value:w \__int_eval:w (#1#2#3 - 62) / 125 ;
15761       #1#2#3
15762     }
15763 \cs_new:Npn \__fp_trig_large_auxviii:ww #1;
15764     {
15765       + #1
15766       \if_int_odd:w #1 \exp_stop_f:
15767         \exp_after:wN \__fp_trig_large_auxix:Nw
15768         \exp_after:wN -
15769       \else:
15770         \exp_after:wN \__fp_trig_large_auxix:Nw
15771         \exp_after:wN +
15772       \fi:
15773     }
15774 \cs_new:Npn \__fp_trig_large_auxix:Nw
15775     {
15776       \exp_after:wN \__fp_use_i_until_s:nw
15777       \exp_after:wN \__fp_trig_large_auxxi:w
15778       \__int_value:w \__int_eval:w \c__fp_leading_shift_int
15779         \prg_replicate:nn { \c_thirteen }
15780           { \__fp_trig_large_auxx:wNNNNN }
15781         + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
15782       ;
15783     }
```

695

```
15784 \cs_new:Npn \__fp_trig_large_auxx:wNNNNN #1; #2 #3#4#5#6
15785   {
15786     \exp_after:wN \__fp_trig_large_pack:NNNNNw
15787     \__int_value:w \__int_eval:w \c__fp_middle_shift_int
15788       #2 \c_eight * #3#4#5#6
15789       #1; #2
15790   }
15791 \cs_new:Npn \__fp_trig_large_auxxi:w #1;
15792   {
15793     \exp_after:wN \__fp_ep_mul_raw:wwwwN
15794     \__int_value:w \__int_eval:w \c_zero \__fp_ep_to_ep_loop:N #1 ; ; !
15795     0,{7853}{9816}{3397}{4483}{0961}{5661};
15796     \__fp_trig_small:ww
15797   }
```

(*End definition for* `\__fp_trig_large_auxvii:w` *and others.*)

30.1.6 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:wwN` or `\__fp_ep_inv_to_-float: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 will control 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.

```
15798 \cs_new:Npn \__fp_sin_series_o:NNwwww #1#2#3. #4;
15799   {
15800     \__fp_fixed_mul:wwn #4; #4;
15801       {
15802         \exp_after:wN \__fp_sin_series_aux_o:NNnwww
15803         \exp_after:wN #1
15804         \__int_value:w
15805           \if_int_odd:w \__int_eval:w (#3 + \c_two) / \c_four \__int_eval_end:
15806             #2
```

696

```
15807              \else:
15808                \if_meaning:w #2 0 2 \else: 0 \fi:
15809            \fi:
15810          {#3}
15811        }
15812      }
15813 \cs_new:Npn \__fp_sin_series_aux_o:NNnwww #1#2#3 #4; #5,#6;
15814    {
15815      \if_int_odd:w \__int_eval:w #3 / \c_two \__int_eval_end:
15816        \exp_after:wN \use_i:nn
15817      \else:
15818        \exp_after:wN \use_ii:nn
15819      \fi:
15820      { % 1/18!
15821        \__fp_fixed_mul_sub_back:wwwn     {0000}{0000}{0000}{0001}{5619}{2070};
15822                                     #4;{0000}{0000}{0000}{0477}{9477}{3324};
15823        \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0011}{4707}{4559}{7730};
15824        \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{2087}{6756}{9878}{6810};
15825        \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0027}{5573}{1922}{3985}{8907};
15826        \__fp_fixed_mul_sub_back:wwwn #4;{0000}{2480}{1587}{3015}{8730}{1587};
15827        \__fp_fixed_mul_sub_back:wwwn #4;{0013}{8888}{8888}{8888}{8888}{8889};
15828        \__fp_fixed_mul_sub_back:wwwn #4;{0416}{6666}{6666}{6666}{6666}{6667};
15829        \__fp_fixed_mul_sub_back:wwwn #4;{5000}{0000}{0000}{0000}{0000}{0000};
15830        \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
15831        { \__fp_fixed_continue:wn 0, }
15832      }
15833      { % 1/17!
15834        \__fp_fixed_mul_sub_back:wwwn     {0000}{0000}{0000}{0028}{1145}{7254};
15835                                     #4;{0000}{0000}{0000}{7647}{1637}{3182};
15836        \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0160}{5904}{3836}{8216};
15837        \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0002}{5052}{1083}{8544}{1719};
15838        \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0275}{5731}{9223}{9858}{9065};
15839        \__fp_fixed_mul_sub_back:wwwn #4;{0001}{9841}{2698}{4126}{9841}{2698};
15840        \__fp_fixed_mul_sub_back:wwwn #4;{0083}{3333}{3333}{3333}{3333}{3333};
15841        \__fp_fixed_mul_sub_back:wwwn #4;{1666}{6666}{6666}{6666}{6666}{6667};
15842        \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
15843        { \__fp_ep_mul:wwwwn 0, } #5,#6;
15844      }
15845      {
15846        \exp_after:wN \__fp_sanitize:Nw
15847        \exp_after:wN #2
15848        \__int_value:w \__int_eval:w #1
15849      }
15850      #2
15851    }
```

(*End 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`

697

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

$$\tan(x) \simeq \frac{x(1 - x^2(a_1 - x^2(a_2 - x^2(a_3 - x^2(a_4 - x^2a_5)))))}{1 - x^2(b_1 - x^2(b_2 - x^2(b_3 - x^2(b_4 - x^2b_5))))}.$$

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.

```
15852 \cs_new:Npn \__fp_tan_series_o:NNwwww #1#2#3. #4;
15853   {
15854     \__fp_fixed_mul:wwn #4; #4;
15855       {
15856         \exp_after:wN \__fp_tan_series_aux_o:Nnwww
15857         \__int_value:w
15858           \if_int_odd:w \__int_eval:w #3 / \c_two \__int_eval_end:
15859             \exp_after:wN \reverse_if:N
15860           \fi:
15861           \if_meaning:w #1#2 2 \else: 0 \fi:
15862         {#3}
15863       }
15864   }
15865 \cs_new:Npn \__fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5;
15866   {
15867     \__fp_fixed_mul_sub_back:wwwn     {0000}{0000}{1527}{3493}{0856}{7059};
15868                                   #3; {0000}{0159}{6080}{0274}{5257}{6472};
15869     \__fp_fixed_mul_sub_back:wwwn #3; {0002}{4571}{2320}{0157}{2558}{8481};
15870     \__fp_fixed_mul_sub_back:wwwn #3; {0115}{5830}{7533}{5397}{3168}{2147};
15871     \__fp_fixed_mul_sub_back:wwwn #3; {1929}{8245}{6140}{3508}{7719}{2982};
15872     \__fp_fixed_mul_sub_back:wwwn #3;{10000}{0000}{0000}{0000}{0000}{0000};
15873     { \__fp_ep_mul:wwwwn 0, } #4,#5;
15874       {
15875         \__fp_fixed_mul_sub_back:wwwn     {0000}{0007}{0258}{0681}{9408}{4706};
15876                                       #3;{0000}{2343}{7175}{1399}{6151}{7670};
15877         \__fp_fixed_mul_sub_back:wwwn #3;{0019}{2638}{4588}{9232}{8861}{3691};
15878         \__fp_fixed_mul_sub_back:wwwn #3;{0536}{6357}{0691}{4344}{6852}{4252};
15879         \__fp_fixed_mul_sub_back:wwwn #3;{5263}{1578}{9473}{6842}{1052}{6315};
15880         \__fp_fixed_mul_sub_back:wwwn#3;{10000}{0000}{0000}{0000}{0000}{0000};
15881           {
15882             \reverse_if:N \if_int_odd:w
15883                 \__int_eval:w (#2 - \c_one) / \c_two \__int_eval_end:
15884               \exp_after:wN \__fp_reverse_args:Nww
15885             \fi:
15886             \__fp_ep_div:wwwwn 0,
15887           }
15888       }
15889       {
15890         \exp_after:wN \__fp_sanitize:Nw
15891         \exp_after:wN #1
15892         \__int_value:w \__int_eval:w \__fp_ep_to_float:wwN
```

```
15893        }
15894          #1
15895        }
```

(*End definition for* `\__fp_tan_series_o:NNwwww` *and* `\__fp_tan_series_aux_o:Nnwww`.)

30.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arc-cosecant, 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 $\mathrm{atan}(y,x) = \mathrm{atan}(y/x)$ 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

$$\mathrm{acos}\, x = \mathrm{atan}(\sqrt{1-x^2}, x) \tag{5}$$

$$\mathrm{asin}\, x = \mathrm{atan}(x, \sqrt{1-x^2}) \tag{6}$$

$$\mathrm{asec}\, x = \mathrm{atan}(\sqrt{x^2-1}, 1) \tag{7}$$

$$\mathrm{acsc}\, x = \mathrm{atan}(1, \sqrt{x^2-1}) \tag{8}$$

$$\mathrm{atan}\, x = \mathrm{atan}(x, 1) \tag{9}$$

$$\mathrm{acot}\, x = \mathrm{atan}(1, x). \tag{10}$$

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 $\mathrm{atan}(y,x) = \mathrm{atan}(y/x)$ is the angular coordinate of the point (x, y).

As for direct trigonometric functions, the first step in computing $\mathrm{atan}(y,x)$ is argument reduction. The sign of y will give 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 $\mathrm{atan}\frac{|y|}{x}$ is given by a nicely convergent Taylor series;

1 $0 < 0.41421x < |y| < x$, then $\mathrm{atan}\frac{|y|}{x} = \frac{\pi}{4} - \mathrm{atan}\frac{x-|y|}{x+|y|}$;

2 $0 < 0.41421|y| < x < |y|$, then $\mathrm{atan}\frac{|y|}{x} = \frac{\pi}{4} + \mathrm{atan}\frac{-x+|y|}{x+|y|}$;

3 $0 < x < 0.41421|y|$, then $\mathrm{atan}\frac{|y|}{x} = \frac{\pi}{2} - \mathrm{atan}\frac{x}{|y|}$;

4 $0 < -x < 0.41421|y|$, then $\mathrm{atan}\frac{|y|}{x} = \frac{\pi}{2} + \mathrm{atan}\frac{-x}{|y|}$;

5 $0 < 0.41421|y| < -x < |y|$, then $\mathrm{atan}\frac{|y|}{x} = \frac{3\pi}{4} - \mathrm{atan}\frac{x+|y|}{-x+|y|}$;

6 $0 < -0.41421x < |y| < -x$, then $\mathrm{atan}\frac{|y|}{x} = \frac{3\pi}{4} + \mathrm{atan}\frac{-x-|y|}{-x+|y|}$;

7 $0 < |y| < -0.41421x$, then $\operatorname{atan} \frac{|y|}{x} = \pi - \operatorname{atan} \frac{|y|}{-x}$.

In the following, we will 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.

30.2.1 Arctangent and arccotangent

__fp_atan_o:Nw
__fp_acot_o:Nw
__fp_atan_dispatch_o:NNnNw

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. Here, we dispatch according to the number of arguments. The one-argument versions of arctangent and arccotangent are special cases of the two-argument ones: $\operatorname{atan}(y) = \operatorname{atan}(y, 1) = \operatorname{acot}(1, y)$ and $\operatorname{acot}(x) = \operatorname{atan}(1, x) = \operatorname{acot}(x, 1)$.

```
15896 \cs_new:Npn \__fp_atan_o:Nw
15897   {
15898     \__fp_atan_dispatch_o:NNnNw
15899       \__fp_acotii_o:Nww \__fp_atanii_o:Nww { atan }
15900   }
15901 \cs_new:Npn \__fp_acot_o:Nw
15902   {
15903     \__fp_atan_dispatch_o:NNnNw
15904       \__fp_atanii_o:Nww \__fp_acotii_o:Nww { acot }
15905   }
15906 \cs_new:Npn \__fp_atan_dispatch_o:NNnNw #1#2#3#4#5@
15907   {
15908     \if_case:w
15909       \__int_eval:w \__fp_array_count:n {#5} - \c_one \__int_eval_end:
15910         \exp_after:wN #1 \exp_after:wN #4 \c_one_fp #5
15911         \exp:w
15912     \or: #2 #4 #5 \exp:w
15913     \else:
15914       \__msg_kernel_expandable_error:nnnnn
15915         { kernel } { fp-num-args } { #3() } { 1 } { 2 }
15916       \exp_after:wN \c_nan_fp \exp:w
15917     \fi:
15918     \exp_after:wN \exp_end:
15919   }
```

(*End definition for* __fp_atan_o:Nw, __fp_acot_o:Nw, *and* __fp_atan_dispatch_o:NNnNw.)

__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 $\{\pm\pi/4, \pm3\pi/4\}$ (in degrees, $\{\pm45, \pm135\}$). 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 $\{\pm0, \pm\pi\}$ (in degrees, $\{\pm0, \pm180\}$). Since $\operatorname{acot}(x, y) = \operatorname{atan}(y, x)$, `\__fp_acotii_o:ww` simply reverses its two arguments.

```
15920 \cs_new:Npn \__fp_atanii_o:Nww
15921     #1 \s__fp \__fp_chk:w #2#3#4; \s__fp \__fp_chk:w #5
15922   {
15923     \if_meaning:w 3 #2 \__fp_case_return_i_o:ww \fi:
15924     \if_meaning:w 3 #5 \__fp_case_return_ii_o:ww \fi:
15925     \if_case:w
```

```
15926        \if_meaning:w #2 #5
15927          \if_meaning:w 1 #2 \c_ten \else: \c_zero \fi:
15928        \else:
15929          \if_int_compare:w #2 > #5 \c_one \else: \c_two \fi:
15930        \fi:
15931          \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 \c_two }
15932      \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 \c_four }
15933      \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 \c_zero }
15934      \fi:
15935      \__fp_atan_normal_o:NNnwNnw #1
15936      \s__fp \__fp_chk:w #2#3#4;
15937      \s__fp \__fp_chk:w #5
15938    }
15939 \cs_new:Npn \__fp_acotii_o:Nww #1#2; #3;
15940    { \__fp_atanii_o:Nww #1#3; #2; }
```

(*End 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; 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, atan z will be computed to be 0, and the result will be $[#3/2] \cdot \pi/4$ if the sign #5 of x is positive, and $[(7 - #3)/2] \cdot \pi/4$ for negative x, where the divisions are rounded up.

```
15941 \cs_new:Npn \__fp_atan_inf_o:NNNw #1#2#3 \s__fp \__fp_chk:w #4#5#6;
15942    {
15943      \exp_after:wN \__fp_atan_combine_o:NwwwwwN
15944      \exp_after:wN #2
15945      \__int_value:w \__int_eval:w
15946        \if_meaning:w 2 #5 \c_seven - \fi: #3 \exp_after:wN ;
15947      \c__fp_one_fixed_tl ;
15948      {0000}{0000}{0000}{0000}{0000}{0000};
15949      0,{0000}{0000}{0000}{0000}{0000}{0000}; #1
15950    }
```

(*End 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 atan$(x, \sqrt{1 - x^2})$ without intermediate rounding errors.

```
15951 \cs_new_protected:Npn \__fp_atan_normal_o:NNnwNnw
15952    #1 \s__fp \__fp_chk:w 1#2#3#4; \s__fp \__fp_chk:w 1#5#6#7;
15953    {
15954      \__fp_atan_test_o:NwwNwwN
15955        #2 #3, #4{0000}{0000};
15956        #5 #6, #7{0000}{0000}; #1
15957    }
```

(*End 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 what z will be, so we simply leave a contribution to the octant: $\langle octant \rangle \to 7 - \langle octant \rangle$ 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.

```
15958 \cs_new:Npn \__fp_atan_test_o:NwwNwwN #1#2,#3; #4#5,#6;
15959   {
15960     \exp_after:wN \__fp_atan_combine_o:NwwwwwN
15961     \exp_after:wN #1
15962     \__int_value:w \__int_eval:w
15963       \if_meaning:w 2 #4
15964         \c_seven - \__int_eval:w
15965       \fi:
15966       \if_int_compare:w
15967           \__fp_ep_compare:wwww #2,#3; #5,#6; > \c_zero
15968         \c_three -
15969         \exp_after:wN \__fp_reverse_args:Nww
15970       \fi:
15971       \__fp_atan_div:wnwwnw #2,#3; #5,#6;
15972   }
```

(*End 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 $\operatorname{atan} \frac{b-a}{a+b}$. Otherwise, the octant is 0 (again, combined with earlier terms) and we wish to compute $\operatorname{atan} \frac{a}{b}$. In any case, call `\__fp_atan_auxi:ww` followed by z, as a comma-delimited exponent and a fixed point number.

```
15973 \cs_new:Npn \__fp_atan_div:wnwwnw #1,#2#3; #4,#5#6;
15974   {
15975     \if_int_compare:w
15976       \__int_eval:w 41421 * #5 < #2 000
15977         \if_case:w \__int_eval:w #4 - #1 \__int_eval_end: 00 \or: 0 \fi:
15978       \exp_stop_f:
15979       \exp_after:wN \__fp_atan_near:wwwn
15980     \fi:
15981     \c_zero
15982     \__fp_ep_div:wwwwn #1,{#2}#3; #4,{#5}#6;
15983     \__fp_atan_auxi:ww
15984   }
15985 \cs_new:Npn \__fp_atan_near:wwwn
15986     \c_zero \__fp_ep_div:wwwwn #1,#2; #3,
15987   {
15988     \c_one
15989     \__fp_ep_to_fixed:wwn #1 - #3, #2;
```

```
15990        \__fp_atan_near_aux:wwn
15991      }
15992   \cs_new:Npn \__fp_atan_near_aux:wwn #1; #2;
15993      {
15994        \__fp_fixed_add:wwn #1; #2;
15995        { \__fp_fixed_sub:wwn #2; #1; { \__fp_ep_div:wwwwn 0, } 0, }
15996      }
```

(*End definition for* __fp_atan_div:wnwwnw, __fp_atan_near:wwwn, *and* __fp_atan_near_aux:wwn.)

__fp_atan_auxi:ww   Convert z from a representation as an exponent and a fixed point number in $[0.01, 1)$ to a
__fp_atan_auxii:w   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.

```
15997   \cs_new:Npn \__fp_atan_auxi:ww #1,#2;
15998      { \__fp_ep_to_fixed:wwn #1,#2; \__fp_atan_auxii:w #1,#2; }
15999   \cs_new:Npn \__fp_atan_auxii:w #1;
16000      {
16001        \__fp_fixed_mul:wwn #1; #1;
16002        {
16003          \__fp_atan_Taylor_loop:www 39 ;
16004            {0000}{0000}{0000}{0000}{0000}{0000} ;
16005        }
16006        ! #1;
16007      }
```

(*End definition for* __fp_atan_auxi:ww *and* __fp_atan_auxii:w.)

__fp_atan_Taylor_loop:www   We compute the series of $(\operatorname{atan} z)/z$. A typical intermediate stage has #1 $= 2k-1$,
__fp_atan_Taylor_break:w   #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.

```
16008   \cs_new:Npn \__fp_atan_Taylor_loop:www #1; #2; #3;
16009      {
16010        \if_int_compare:w #1 = \c_minus_one
16011          \__fp_atan_Taylor_break:w
16012        \fi:
16013        \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl ; #1;
16014        \__fp_rrot:www \__fp_fixed_mul_sub_back:wwwn #2; #3;
16015        {
16016          \exp_after:wN \__fp_atan_Taylor_loop:www
16017          \__int_value:w \__int_eval:w #1 - \c_two ;
16018        }
16019        #3;
16020      }
16021   \cs_new:Npn \__fp_atan_Taylor_break:w
16022      \fi: #1 \__fp_fixed_mul_sub_back:wwwn #2; #3 !
16023      { \fi: ; #2 ; }
```

(*End definition for* __fp_atan_Taylor_loop:www *and* __fp_atan_Taylor_break:w.)

__fp_atan_combine_o:NwwwwwN   This receives a ⟨*sign*⟩, an ⟨*octant*⟩, a fixed point value of $(\operatorname{atan} z)/z$, a fixed point num-
__fp_atan_combine_aux:ww   ber z, and another representation of z, as an ⟨*exponent*⟩ and the fixed point number

$10^{-\langle exponent \rangle} z$, 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

$$\langle sign \rangle \left(\left\lceil \frac{\langle octant \rangle}{2} \right\rceil \frac{\pi}{4} + (-1)^{\langle octant \rangle} \frac{\operatorname{atan} z}{z} \cdot z \right) , \tag{11}$$

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 · #4. In both cases, convert to a floating point with `\__fp_fixed_to_-float:wN`.

```
16024 \cs_new:Npn \__fp_atan_combine_o:NwwwwwN #1 #2; #3; #4; #5,#6; #7
16025   {
16026     \exp_after:wN \__fp_sanitize:Nw
16027     \exp_after:wN #1
16028     \__int_value:w \__int_eval:w
16029       \if_meaning:w 0 #2
16030         \exp_after:wN \use_i:nn
16031       \else:
16032         \exp_after:wN \use_ii:nn
16033       \fi:
16034       { #5 \__fp_fixed_mul:wwn #3; #6; }
16035       {
16036         \__fp_fixed_mul:wwn #3; #4;
16037         {
16038           \exp_after:wN \__fp_atan_combine_aux:ww
16039           \__int_value:w \__int_eval:w #2 / \c_two ; #2;
16040         }
16041       }
16042       { #7 \__fp_fixed_to_float:wN \__fp_fixed_to_float_rad:wN }
16043       #1
16044   }
16045 \cs_new:Npn \__fp_atan_combine_aux:ww #1; #2;
16046   {
16047     \__fp_fixed_mul_short:wwn
16048       {7853}{9816}{3397}{4483}{0961}{5661};
16049       {#1}{0000}{0000};
16050     {
16051       \if_int_odd:w #2 \exp_stop_f:
16052         \exp_after:wN \__fp_fixed_sub:wwn
16053       \else:
16054         \exp_after:wN \__fp_fixed_add:wwn
16055       \fi:
16056     }
16057   }
```

(*End definition for* `\__fp_atan_combine_o:NwwwwwN` *and* `\__fp_atan_combine_aux:ww`.)

30.2.2 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

704

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).

```
16058 \cs_new:Npn \__fp_asin_o:w #1 \s__fp \__fp_chk:w #2#3; @
16059   {
16060     \if_case:w #2 \exp_stop_f:
16061       \__fp_case_return_same_o:w
16062     \or:
16063       \__fp_case_use:nw
16064         { \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { asin } { asind } } }
16065     \or:
16066       \__fp_case_use:nw
16067         { \__fp_invalid_operation_o:fw { #1 { asin } { asind } } }
16068     \else:
16069       \__fp_case_return_same_o:w
16070     \fi:
16071     \s__fp \__fp_chk:w #2 #3;
16072   }
```

(End 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.

```
16073 \cs_new:Npn \__fp_acos_o:w #1 \s__fp \__fp_chk:w #2#3; @
16074   {
16075     \if_case:w #2 \exp_stop_f:
16076       \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 \c_four }
16077     \or:
16078       \__fp_case_use:nw
16079         {
16080           \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { acos } { acosd } }
16081             \__fp_reverse_args:Nww
16082         }
16083     \or:
16084       \__fp_case_use:nw
16085         { \__fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
16086     \else:
16087       \__fp_case_return_same_o:w
16088     \fi:
16089     \s__fp \__fp_chk:w #2 #3;
16090   }
```

(End definition for `\__fp_acos_o:w`.*)*

`\__fp_asin_normal_o:NfwNnnnnw`   If the exponent #5 is strictly less than 1, the operand lies within $(-1, 1)$ and the operation is permitted: call `\__fp_asin_auxi_o:nNww` 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:nNww`. 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.

```
16091 \cs_new:Npn \__fp_asin_normal_o:NfwNnnnnw
```

```
16092        #1#2#3 \s__fp \__fp_chk:w 1#4#5#6#7#8#9;
16093    {
16094      \if_int_compare:w #5 < \c_one
16095        \exp_after:wN \__fp_use_none_until_s:w
16096      \fi:
16097      \if_int_compare:w \__int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
16098        \exp_after:wN \__fp_use_none_until_s:w
16099      \fi:
16100      \__fp_use_i:ww
16101      \__fp_invalid_operation_o:fw {#2}
16102        \s__fp \__fp_chk:w 1#4{#5}{#6}{#7}{#8}{#9};
16103      \__fp_asin_auxi_o:NnNww
16104        #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000};
16105    }
```

(*End definition for* __fp_asin_normal_o:NfwNnnnnw.)

__fp_asin_auxi_o:NnNww We compute $x/\sqrt{1-x^2}$. This function is used by `asin` and `acos`, but also by `acsc` and
__fp_asin_isqrt:wn `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`.

```
16106  \cs_new:Npn \__fp_asin_auxi_o:NnNww #1#2#3#4,#5;
16107    {
16108      \__fp_ep_to_fixed:wwn #4,#5;
16109      \__fp_asin_isqrt:wn
16110      \__fp_ep_mul:wwwwn #4,#5;
16111      \__fp_ep_to_ep:wwN
16112      \__fp_fixed_continue:wn
16113      { #2 \__fp_atan_test_o:NwwNwwN #3 }
16114      0 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1
16115    }
16116  \cs_new:Npn \__fp_asin_isqrt:wn #1;
16117    {
16118      \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl ; #1;
16119      {
16120        \__fp_fixed_add_one:wN #1;
16121        \__fp_fixed_continue:wn { \__fp_ep_mul:wwwwn 0, } 0,
16122      }
16123      \__fp_ep_isqrt:wwn
16124    }
```

(*End definition for* __fp_asin_auxi_o:NnNww *and* __fp_asin_isqrt:wn.)

30.2.3 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.

```
16125 \cs_new:Npn \__fp_acsc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
16126   {
16127     \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
16128           \__fp_case_use:nw
16129             { \__fp_invalid_operation_o:fw { #1 { acsc } { acscd } } }
16130     \or:    \__fp_case_use:nw
16131             { \__fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acscd } } } }
16132     \or:    \__fp_case_return_o:Nw \c_zero_fp
16133     \or:    \__fp_case_return_same_o:w
16134     \else: \__fp_case_return_o:Nw \c_minus_zero_fp
16135     \fi:
16136     \s__fp \__fp_chk:w #2 #3 #4;
16137   }
```

(*End 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.

```
16138 \cs_new:Npn \__fp_asec_o:w #1 \s__fp \__fp_chk:w #2#3; @
16139   {
16140     \if_case:w #2 \exp_stop_f:
16141       \__fp_case_use:nw
16142         { \__fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
16143     \or:
16144       \__fp_case_use:nw
16145         {
16146           \__fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
16147             \__fp_reverse_args:Nww
16148         }
16149     \or:    \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 \c_four }
16150     \else: \__fp_case_return_same_o:w
16151     \fi:
16152     \s__fp \__fp_chk:w #2 #3;
16153   }
```

(*End 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:nWww (with all the appropriate arguments). This computes what we want thanks to $\mathrm{acsc}(x) = \mathrm{asin}(1/x)$ and $\mathrm{asec}(x) = \mathrm{acos}(1/x)$.

```
16154 \cs_new:Npn \__fp_acsc_normal_o:NfwNnw #1#2#3 \s__fp \__fp_chk:w 1#4#5#6;
16155   {
16156     \int_compare:nNnTF {#5} < \c_one
16157       {
16158         \__fp_invalid_operation_o:fw {#2}
16159           \s__fp \__fp_chk:w 1#4{#5}#6;
```

```
16160          }
16161          {
16162            \__fp_ep_div:wwwwn
16163              1,{1000}{0000}{0000}{0000}{0000}{0000};
16164              #5,#6{0000}{0000};
16165            { \__fp_asin_auxi_o:NnNww #1 {#3} #4 }
16166          }
16167      }
```

(*End definition for* `\__fp_acsc_normal_o:NfwNnw`.)

```
16168 ⟨/initex | package⟩
```

31   `l3fp-convert` implementation

```
16169 ⟨*initex | package⟩
```

```
16170 ⟨@@=fp⟩
```

31.1   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 will be the `dot` auxiliary, which removes a trailing dot if any. We then clean-up with the `end` auxiliary, keeping only the number.

```
16171 \cs_new:Npn \__fp_trim_zeros:w #1 ;
16172   {
16173     \__fp_trim_zeros_loop:w #1
16174       ; \__fp_trim_zeros_loop:w 0; \__fp_trim_zeros_dot:w .; \s__stop
16175   }
16176 \cs_new:Npn \__fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
16177 \cs_new:Npn \__fp_trim_zeros_dot:w #1 .; { \__fp_trim_zeros_end:w #1 ; }
16178 \cs_new:Npn \__fp_trim_zeros_end:w #1 ; #2 \s__stop { #1 }
```

(*End definition for* `\__fp_trim_zeros:w` *and others.*)

31.2   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`.

```
16179 \cs_new:Npn \fp_to_scientific:N #1
16180   { \exp_after:wN \__fp_to_scientific_dispatch:w #1 }
16181 \cs_generate_variant:Nn \fp_to_scientific:N { c }
16182 \cs_new:Npn \fp_to_scientific:n
16183   {
16184     \exp_after:wN \__fp_to_scientific_dispatch:w
16185     \exp:w \exp_end_continue_f:w \__fp_parse:n
16186   }
```

(*End definition for* `\fp_to_scientific:N` *and* `\fp_to_scientific:n`. *These functions are documented on page 183.*)

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. Finally trim zeros.

```
16187 \cs_new:Npn \__fp_to_scientific_dispatch:w \s__fp \__fp_chk:w #1#2
16188   {
16189     \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
16190     \if_case:w #1 \exp_stop_f:
16191         \__fp_case_return:nw { 0 }
16192     \or: \exp_after:wN \__fp_to_scientific_normal:wnnnnn
16193     \or:
16194       \__fp_case_use:nw
16195         {
16196           \__fp_invalid_operation:nnw
16197             {
16198               \exp_after:wN 1
16199               \exp_after:wN e
16200               \int_use:N \c__fp_max_exponent_int
16201             }
16202           { fp_to_scientific }
16203         }
16204     \or:
16205       \__fp_case_use:nw
16206         {
16207           \__fp_invalid_operation:nnw
16208             { 0 }
16209             { fp_to_scientific }
16210         }
16211     \fi:
16212     \s__fp \__fp_chk:w #1 #2
16213   }
16214 \cs_new:Npn \__fp_to_scientific_normal:wnnnnn
16215   \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
16216   {
16217     \if_int_compare:w #2 = \c_one
16218       \exp_after:wN \__fp_to_scientific_normal:wNw
16219     \else:
16220       \exp_after:wN \__fp_to_scientific_normal:wNw
16221       \exp_after:wN e
16222       \__int_value:w \__int_eval:w #2 - \c_one
16223     \fi:
16224     ; #3 #4 #5 #6 ;
16225   }
16226 \cs_new:Npn \__fp_to_scientific_normal:wNw #1 ; #2#3;
16227   { \__fp_trim_zeros:w #2.#3 ; #1 }
```

(*End definition for* __fp_to_scientific_dispatch:w, __fp_to_scientific_normal:wnnnnn, *and* __-
fp_to_scientific_normal:wNw.)

31.3 Decimal representation

All three public variants are based on the same `\__fp_to_decimal_dispatch:w` after evaluating their argument to an internal floating point.

```
16228 \cs_new:Npn \fp_to_decimal:N #1
16229   { \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
16230 \cs_generate_variant:Nn \fp_to_decimal:N { c }
16231 \cs_new:Npn \fp_to_decimal:n
16232   {
16233     \exp_after:wN \__fp_to_decimal_dispatch:w
16234     \exp:w \exp_end_continue_f:w \__fp_parse:n
16235   }
```

(*End definition for* `\fp_to_decimal:N` *and* `\fp_to_decimal:n`. *These functions are documented on page* *182*.)

The structure is similar to `\__fp_to_scientific_dispatch:w`. Insert - for negative numbers. Zero gives 0, ±∞ and NaN yield an "invalid operation" exception; note that ±∞ 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.

```
16236 \cs_new:Npn \__fp_to_decimal_dispatch:w \s__fp \__fp_chk:w #1#2
16237   {
16238     \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
16239     \if_case:w #1 \exp_stop_f:
16240         \__fp_case_return:nw { 0 }
16241     \or: \exp_after:wN \__fp_to_decimal_normal:wnnnnn
16242     \or:
16243       \__fp_case_use:nw
16244         {
16245           \__fp_invalid_operation:nnw
16246             {
16247               \exp_after:wN \exp_after:wN \exp_after:wN 1
16248               \prg_replicate:nn \c__fp_max_exponent_int 0
16249             }
16250           { fp_to_decimal }
16251         }
16252     \or:
16253       \__fp_case_use:nw
16254         {
16255           \__fp_invalid_operation:nnw
16256             { 0 }
16257           { fp_to_decimal }
16258         }
16259     \fi:
16260     \s__fp \__fp_chk:w #1 #2
16261   }
16262 \cs_new:Npn \__fp_to_decimal_normal:wnnnnn
16263   \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
16264   {
16265     \int_compare:nNnTF {#2} > \c_zero
```

The left margin labels:

`\fp_to_decimal:N`
`\fp_to_decimal:c`
`\fp_to_decimal:n`

`\__fp_to_decimal_dispatch:w`
`\__fp_to_decimal_normal:wnnnnn`
`\__fp_to_decimal_large:Nnnw`
`\__fp_to_decimal_huge:wnnnn`

```
16266        {
16267          \int_compare:nNnTF {#2} < \c_sixteen
16268            {
16269              \__fp_decimate:nNnnnn { \c_sixteen - #2 }
16270                \__fp_to_decimal_large:Nnnw
16271            }
16272            {
16273              \exp_after:wN \exp_after:wN
16274              \exp_after:wN \__fp_to_decimal_huge:wnnnn
16275              \prg_replicate:nn { #2 - \c_sixteen } { 0 } ;
16276            }
16277          {#3} {#4} {#5} {#6}
16278        }
16279        {
16280          \exp_after:wN \__fp_trim_zeros:w
16281          \exp_after:wN 0
16282          \exp_after:wN .
16283          \exp:w \exp_end_continue_f:w \prg_replicate:nn { - #2 } { 0 }
16284          #3#4#5#6 ;
16285        }
16286    }
16287 \cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4;
16288    {
16289      \exp_after:wN \__fp_trim_zeros:w \__int_value:w
16290        \if_int_compare:w #2 > \c_zero
16291          #2
16292        \fi:
16293        \exp_stop_f:
16294        #3.#4 ;
16295    }
16296 \cs_new:Npn \__fp_to_decimal_huge:wnnnn #1; #2#3#4#5 { #2#3#4#5 #1 }
```

(*End definition for* `\__fp_to_decimal_dispatch:w` *and others.*)

31.4   Token list representation

\fp_to_tl:N These three public functions evaluate their argument, then pass it to `\__fp_to_tl_-`
\fp_to_tl:c `dispatch:w`.
\fp_to_tl:n

```
16297 \cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \__fp_to_tl_dispatch:w #1 }
16298 \cs_generate_variant:Nn \fp_to_tl:N { c }
16299 \cs_new:Npn \fp_to_tl:n
16300    {
16301      \exp_after:wN \__fp_to_tl_dispatch:w
16302      \exp:w \exp_end_continue_f:w \__fp_parse:n
16303    }
```

(*End definition for* `\fp_to_tl:N` *and* `\fp_to_tl:n`*. These functions are documented on page 183.*)

__fp_to_tl_dispatch:w   A structure similar to `\__fp_to_scientific_dispatch:w` and `\__fp_to_decimal_-`
__fp_to_tl_normal:nnnnn   `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.

```
16304 \cs_new:Npn \__fp_to_tl_dispatch:w \s__fp \__fp_chk:w #1#2
16305    {
```

```
16306        \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
16307        \if_case:w #1 \exp_stop_f:
16308            \__fp_case_return:nw { 0 }
16309        \or:   \exp_after:wN \__fp_to_tl_normal:nnnnn
16310        \or:   \__fp_case_return:nw { inf }
16311        \else: \__fp_case_return:nw { nan }
16312        \fi:
16313    }
16314 \cs_new:Npn \__fp_to_tl_normal:nnnnn #1
16315    {
16316        \if_int_compare:w #1 > \c_sixteen
16317            \exp_after:wN \__fp_to_scientific_normal:wnnnnn
16318        \else:
16319            \if_int_compare:w #1 < - \c_two
16320                \exp_after:wN \exp_after:wN
16321                \exp_after:wN \__fp_to_scientific_normal:wnnnnn
16322            \else:
16323                \exp_after:wN \exp_after:wN
16324                \exp_after:wN \__fp_to_decimal_normal:wnnnnn
16325            \fi:
16326        \fi:
16327        \s__fp \__fp_chk:w 1 0 {#1}
16328    }
```

(*End definition for* __fp_to_tl_dispatch:w *and* __fp_to_tl_normal:nnnnn.)

31.5   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.

31.6   Convert to dimension or integer

\fp_to_dim:N   These three public functions rely on \fp_to_decimal:n internally. We make sure to
\fp_to_dim:c   produce pt with category other.
\fp_to_dim:n
```
16329 \cs_new:Npn \fp_to_dim:N #1
16330    { \fp_to_decimal:N #1 pt }
16331 \cs_generate_variant:Nn \fp_to_dim:N { c }
16332 \cs_new:Npn \fp_to_dim:n #1
16333    { \fp_to_decimal:n {#1} pt }
```

(*End definition for* \fp_to_dim:N *and* \fp_to_dim:n. *These functions are documented on page 182.*)

\fp_to_int:N   These three public functions evaluate their argument, then pass it to \fp_to_int_-
\fp_to_int:c   dispatch:w.
\fp_to_int:n
```
16334 \cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \__fp_to_int_dispatch:w #1 }
16335 \cs_generate_variant:Nn \fp_to_int:N { c }
16336 \cs_new:Npn \fp_to_int:n
16337    {
16338        \exp_after:wN \__fp_to_int_dispatch:w
16339        \exp:w \exp_end_continue_f:w \__fp_parse:n
16340    }
```

(End definition for `\fp_to_int:N` *and* `\fp_to_int:n`*. These functions are documented on page 183.)*

`\__fp_to_int_dispatch:w` 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 will be no trailing dot nor zero.

```
16341 \cs_new:Npn \__fp_to_int_dispatch:w #1;
16342   {
16343     \exp_after:wN \__fp_to_decimal_dispatch:w \exp:w \exp_end_continue_f:w
16344       \__fp_round:Nwn \__fp_round_to_nearest:NNN #1; { 0 }
16345   }
```

(End definition for `\__fp_to_int_dispatch:w`*.)*

31.7 Convert from a dimension

`\dim_to_fp:n` The dimension expression (which can in fact be a glue expression) is evaluated, con-
`\__fp_from_dim_test:ww` verted to a number (*i.e.*, expressed in scaled points), then multiplied by $2^{-16} =$
`\__fp_from_dim:wNw` 0.0000152587890625 to give a value expressed in points. The auxiliary `\__fp_mul_-`
`\__fp_from_dim:wNNnnnnnn` `npos_o:Nww` expects the desired ⟨*final sign*⟩ and two floating point operands (of the form
`\__fp_from_dim:wnnnnwNw` `\s__fp ... ;`) as arguments. This set of functions is also used to convert dimension reg-
isters 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:` in `\dim_to_fp:n`.

```
16346 \cs_new:Npn \dim_to_fp:n #1
16347   {
16348     \exp_after:wN \__fp_from_dim_test:ww
16349     \exp_after:wN 0
16350     \exp_after:wN ,
16351     \__int_value:w \etex_glueexpr:D #1 ;
16352   }
16353 \cs_new:Npn \__fp_from_dim_test:ww #1, #2
16354   {
16355     \if_meaning:w 0 #2
16356       \__fp_case_return:nw { \exp_after:wN \c_zero_fp }
16357     \else:
16358       \exp_after:wN \__fp_from_dim:wNw
16359       \__int_value:w \__int_eval:w #1 - \c_four
16360         \if_meaning:w - #2
16361           \exp_after:wN , \exp_after:wN 2 \__int_value:w
16362         \else:
16363           \exp_after:wN , \exp_after:wN 0 \__int_value:w #2
16364         \fi:
16365     \fi:
16366   }
16367 \cs_new:Npn \__fp_from_dim:wNw #1,#2#3;
16368   {
16369     \__fp_pack_twice_four:wNNNNNNNN \__fp_from_dim:wNNnnnnnn ;
16370     #3 000 0000 00 {10}987654321; #2 {#1}
16371   }
16372 \cs_new:Npn \__fp_from_dim:wNNnnnnnn #1; #2#3#4#5#6#7#8#9
16373   { \__fp_from_dim:wnnnnwNn #1 {#2#300} {0000} ; }
16374 \cs_new:Npn \__fp_from_dim:wnnnnwNn #1; #2#3#4#5#6; #7#8
```

```
16375    {
16376      \__fp_mul_npos_o:Nww #7
16377        \s__fp \__fp_chk:w 1 #7 {#5} #1 ;
16378        \s__fp \__fp_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} ;
16379        \prg_do_nothing:
16380    }
```

(*End definition for* `\dim_to_fp:n` *and others. These functions are documented on page 82.*)

31.8   Use and eval

`\fp_use:N` Those public functions are simple copies of the decimal conversions.

`\fp_use:c`

`\fp_eval:n`
```
16381 \cs_new_eq:NN \fp_use:N \fp_to_decimal:N
16382 \cs_generate_variant:Nn \fp_use:N { c }
16383 \cs_new_eq:NN \fp_eval:n \fp_to_decimal:n
```

(*End definition for* `\fp_use:N` *and* `\fp_eval:n`*. These functions are documented on page 183.*)

`\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.
```
16384 \cs_new:Npn \fp_abs:n #1
16385   { \fp_to_decimal:n { abs \__fp_parse:n {#1} } }
```

(*End definition for* `\fp_abs:n`*. This function is documented on page 195.*)

`\fp_max:nn` Similar to `\fp_abs:n`, for consistency with `\int_max:nn`, *etc.*

`\fp_min:nn`
```
16386 \cs_new:Npn \fp_max:nn #1#2
16387   { \fp_to_decimal:n { max ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
16388 \cs_new:Npn \fp_min:nn #1#2
16389   { \fp_to_decimal:n { min ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
```

(*End definition for* `\fp_max:nn` *and* `\fp_min:nn`*. These functions are documented on page 196.*)

31.9   Convert an array of floating points to a comma list

`\__fp_array_to_clist:n` Converts an array of floating point numbers to a comma-list. If speed here ends up
`\__fp_array_to_clist_loop:Nw` 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.
```
16390 \cs_new:Npn \__fp_array_to_clist:n #1
16391   {
16392     \tl_if_empty:nF {#1}
16393       {
16394         \__fp_expand:n
16395           {
16396             { \use_ii:nn }
```

714

```
16397                    \__fp_array_to_clist_loop:Nw #1 { ? \__prg_break: } ;
16398                    \__prg_break_point:
16399                  }
16400            }
16401    }
16402 \cs_new:Npx \__fp_array_to_clist_loop:Nw #1#2;
16403   {
16404     \exp_not:N \use_none:n #1
16405     \exp_not:N \exp_after:wN
16406            {
16407     \exp_not:N    \exp_after:wN ,
16408     \exp_not:N    \exp_after:wN \c_space_tl
16409     \exp_not:N    \exp:w
16410     \exp_not:N    \exp_end_continue_f:w
16411     \exp_not:N    \__fp_to_tl_dispatch:w #1 #2 ;
16412            }
16413     \exp_not:N \__fp_array_to_clist_loop:Nw
16414   }
```

(*End definition for* `\__fp_array_to_clist:n` *and* `\__fp_array_to_clist_loop:Nw.`)

```
16415 ⟨/initex | package⟩
```

32 l3fp-random Implementation

```
16416 ⟨*initex | package⟩
```

```
16417 ⟨@@=fp⟩
```

32.1 Random floating point

`\__fp_rand_o:Nw`
`\__fp_rand_o:`

Check that `random` was called without argument. Random numbers are only available if the engine defines a primitive equivalent to pdfTeX's `\pdfuniformdeviate`. A random floating point number in $[0, 1)$, uniformly distributed with equal-spaced steps of 10^{-16} is obtained by combining two integers in $[0, 10^8 - 1]$.

```
16418 \cs_new:Npn \__fp_rand_o:Nw ? #1 @
16419   {
16420     \tl_if_empty:nTF {#1}
16421       { \__fp_rand_o: }
16422       {
16423         \__msg_kernel_expandable_error:nnnnn
16424           { kernel } { fp-num-args } { rand() } { 0 } { 0 }
16425         \exp_after:wN \c_nan_fp
16426       }
16427   }
16428 \cs_if_exist:NTF \pdftex_uniformdeviate:D
16429   {
16430     \cs_new:Npn \__fp_rand_o:
16431       {
16432         \__fp_parse_o:n
16433           {
16434             .
16435             \exp_after:wN \use_none:n
16436             \__int_value:w \__int_eval:w 1 0000 0000 +
16437               \pdftex_uniformdeviate:D 1 0000 0000 \__int_eval_end:
```

715

```
16438              \exp_after:wN \use_none:n
16439              \__int_value:w \__int_eval:w 1 0000 0000 +
16440                \pdftex_uniformdeviate:D 1 0000 0000 \__int_eval_end:
16441            }
16442        }
16443    }
16444    {
16445      \cs_new:Npn \__fp_rand_o:
16446        {
16447          \__msg_kernel_expandable_error:nn { kernel } { fp-no-random }
16448          \exp_after:wN \c_nan_fp
16449        }
16450    }
```

(*End definition for* __fp_rand_o:Nw *and* __fp_rand_o:.)

32.2   Random integer

<div style="float:left">

__fp_randint_o:Nw

__fp_randint_badarg:w

__fp_randint_e:w

__fp_randint_narrow_e:nn

__fp_randint_wide_e:nn

__fp_randint_wide_e:nnn

__fp_randint_wide_e:nnnn

</div>

Enforce that there is one argument (then add first argument 1) or two arguments. Enforce that they are integers in $(-10^{16}, 10^{16})$ and ordered.

When the range is narrow (less than 2×10^9) we can directly use the primitive. Otherwise we use the primitive twice. To ensure a uniform distribution we combine two uniformly distributed integers into a total range greater or equal to our target range and try again if the random number ended up being too big.

The strange +\c_six ensures that the first argument of __fp_randint_wide_-e:nnn is larger than 10^{-8} times the difference between the end-points even if these are near -10^{16} and $+10^{16}$ (their difference is rounded to sixteen digits). Note that we cannot just concatenate the two random numbers provided by the engine as that may give an integer up to (almost) 2×10^{16} which incurs rounding.

```
16451 \cs_new:Npn \__fp_randint_o:Nw ? #1 @
16452   {
16453     \if_case:w
16454       \__int_eval:w \__fp_array_count:n {#1} - \c_one \__int_eval_end:
16455         \exp_after:wN \__fp_randint_e:w \c_one_fp #1
16456     \or: \__fp_randint_e:w #1
16457     \else:
16458       \__msg_kernel_expandable_error:nnnnn
16459         { kernel } { fp-num-args } { randint() } { 1 } { 2 }
16460       \exp_after:wN \c_nan_fp \exp:w
16461     \fi:
16462     \exp_after:wN \exp_end:
16463   }
16464 \cs_new:Npn \__fp_randint_badarg:w \s__fp \__fp_chk:w #1#2#3;
16465   {
16466     \__fp_int:wTF \s__fp \__fp_chk:w #1#2#3;
16467       {
16468         \if_meaning:w 1 #1
16469           \if_int_compare:w
16470             \use_i_delimit_by_q_stop:nw #3 \q_stop > \c_sixteen
16471             \exp_after:wN \c_one
16472           \fi:
16473         \fi:
16474       }
```

```
16475        { \c_one }
16476    }
16477 \cs_new:Npn \__fp_randint_e:w #1; #2;
16478    {
16479      \if_case:w
16480          \__fp_randint_badarg:w #1;
16481          \__fp_randint_badarg:w #2;
16482          \fp_compare:nNnTF { #1; } > { #2; } { \c_one } { \c_zero }
16483        \fp_compare:nNnTF { #1; + 2e9 } < { #2; }
16484          { \__fp_randint_wide_e:nn }
16485          { \__fp_randint_narrow_e:nn }
16486          { #1; } { #2; }
16487      \or:
16488        \__fp_invalid_operation_tl_o:ff
16489          { randint } { \__fp_array_to_clist:n { #1; #2; } } }
16490        \exp:w
16491      \fi:
16492    }
16493 \cs_if_exist:NTF \pdftex_uniformdeviate:D
16494    {
16495      \cs_new:Npn \__fp_randint_narrow_e:nn #1#2
16496        {
16497          \__fp_parse_o:n
16498            {
16499              #1 +
16500              \pdftex_uniformdeviate:D
16501                \int_eval:n { \fp_to_int:n { #2 - #1 } + \c_one }
16502            }
16503          \exp:w
16504        }
16505      \cs_new:Npn \__fp_randint_wide_e:nn #1 #2
16506        {
16507          \exp_args:Nf \__fp_randint_wide_e:nnn
16508            { \fp_to_int:n { ceil ( ( #2 - #1 + \c_six ) * 1e-8 ) } }
16509            {#1} {#2}
16510        }
16511      \cs_new:Npn \__fp_randint_wide_e:nnn #1#2#3
16512        {
16513          \exp_args:Nf \__fp_randint_wide_e:nnnn
16514            {
16515              \__fp_parse:n
16516                {
16517                  #2 +
16518                  \pdftex_uniformdeviate:D #1 e 8 +
16519                  \pdftex_uniformdeviate:D 1 0000 0000
16520                }
16521            }
16522          {#1} {#2} {#3}
16523        }
16524      \cs_new:Npn \__fp_randint_wide_e:nnnn #1#2#3#4
16525        {
16526        \fp_compare:nNnTF {#1} > {#4}
16527          { \__fp_randint_wide_e:nnn {#2} {#3} {#4} }
16528          { \__fp_exp_after_o:w #1 \exp:w }
```

```
16529            }
16530        }
16531        {
16532          \cs_new:Npn \__fp_randint_narrow_e:nn #1#2
16533            {
16534              \__msg_kernel_expandable_error:nn { kernel } { fp-no-random }
16535              \exp_after:wN \c_nan_fp \exp:w
16536            }
16537          \cs_new:Npn \__fp_randint_wide_e:nn #1#2
16538            {
16539              \__msg_kernel_expandable_error:nn { kernel } { fp-no-random }
16540              \exp_after:wN \c_nan_fp \exp:w
16541            }
16542        }
```

(*End definition for* `\__fp_randint_o:Nw` *and others.*)

```
16543  \__msg_kernel_new:nnn { kernel } { fp-no-random }
16544    { Random~numbers~unavailable }
```

```
16545 ⟨/initex | package⟩
```

33 **l3fp-assign** implementation

```
16546 ⟨*initex | package⟩
```

```
16547 ⟨@@=fp⟩
```

33.1 Assigning values

\fp_new:N   Floating point variables are initialized to be $+0$.

```
16548 \cs_new_protected:Npn \fp_new:N #1
16549    { \cs_new_eq:NN #1 \c_zero_fp }
16550 \cs_generate_variant:Nn \fp_new:N {c}
```

(*End definition for* `\fp_new:N`. *This function is documented on page 181.*)

\fp_set:Nn   Simply use `\__fp_parse:n` within various f-expanding assignments.
\fp_set:cn
\fp_gset:Nn
```
16551 \cs_new_protected:Npn \fp_set:Nn   #1#2
16552    { \tl_set:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
16553 \cs_new_protected:Npn \fp_gset:Nn   #1#2
16554    { \tl_gset:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
16555 \cs_new_protected:Npn \fp_const:Nn #1#2
16556    { \tl_const:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
16557 \cs_generate_variant:Nn \fp_set:Nn {c}
16558 \cs_generate_variant:Nn \fp_gset:Nn {c}
16559 \cs_generate_variant:Nn \fp_const:Nn {c}
```
\fp_gset:cn
\fp_const:Nn
\fp_const:cn

(*End definition for* `\fp_set:Nn`, `\fp_gset:Nn`, *and* `\fp_const:Nn`. *These functions are documented on page 181.*)

\fp_set_eq:NN   Copying a floating point is the same as copying the underlying token list.
\fp_set_eq:cN
\fp_set_eq:Nc
```
16560 \cs_new_eq:NN \fp_set_eq:NN   \tl_set_eq:NN
16561 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
16562 \cs_generate_variant:Nn \fp_set_eq:NN  { c , Nc , cc }
16563 \cs_generate_variant:Nn \fp_gset_eq:NN { c , Nc , cc }
```
\fp_set_eq:cc
\fp_gset_eq:NN
\fp_gset_eq:cN
\fp_gset_eq:Nc
\fp_gset_eq:cc

718

(End definition for `\fp_set_eq:NN` *and* `\fp_gset_eq:NN`*. These functions are documented on page 182.)*

`\fp_zero:N` Setting a floating point to zero: copy `\c_zero_fp`.
`\fp_zero:c`
`\fp_gzero:N` 16564 `\cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }`
`\fp_gzero:c` 16565 `\cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }`
16566 `\cs_generate_variant:Nn \fp_zero:N  { c }`
16567 `\cs_generate_variant:Nn \fp_gzero:N { c }`

(End definition for `\fp_zero:N` *and* `\fp_gzero:N`*. These functions are documented on page 181.)*

`\fp_zero_new:N` Set the floating point to zero, or define it if needed.
`\fp_zero_new:c`
`\fp_gzero_new:N` 16568 `\cs_new_protected:Npn \fp_zero_new:N #1`
`\fp_gzero_new:c` 16569 `  { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }`
16570 `\cs_new_protected:Npn \fp_gzero_new:N #1`
16571 `  { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }`
16572 `\cs_generate_variant:Nn \fp_zero_new:N  { c }`
16573 `\cs_generate_variant:Nn \fp_gzero_new:N { c }`

(End definition for `\fp_zero_new:N` *and* `\fp_gzero_new:N`*. These functions are documented on page 181.)*

33.2 Updating values

These match the equivalent functions in l3int and l3skip.

`\fp_add:Nn` For the sake of error recovery we should not simply set #1 to #1 ± (#2): for instance, if #2
`\fp_add:cn` is 0)+2, the parsing error would be raised at the last closing parenthesis rather than at
`\fp_gadd:Nn` the closing parenthesis in the user argument. Thus we evaluate #2 instead of just putting
`\fp_gadd:cn` parentheses. As an optimization we use `\__fp_parse:n` rather than `\fp_eval:n`, which
`\fp_sub:Nn` would convert the result away from the internal representation and back.
`\fp_sub:cn`
`\fp_gsub:Nn` 16574 `\cs_new_protected:Npn \fp_add:Nn  { \__fp_add:NNNn \fp_set:Nn  + }`
`\fp_gsub:cn` 16575 `\cs_new_protected:Npn \fp_gadd:Nn { \__fp_add:NNNn \fp_gset:Nn + }`
`\__fp_add:NNNn` 16576 `\cs_new_protected:Npn \fp_sub:Nn  { \__fp_add:NNNn \fp_set:Nn  - }`
16577 `\cs_new_protected:Npn \fp_gsub:Nn { \__fp_add:NNNn \fp_gset:Nn - }`
16578 `\cs_new_protected:Npn \__fp_add:NNNn #1#2#3#4`
16579 `  { #1 #3 { #3 #2 \__fp_parse:n {#4} } }`
16580 `\cs_generate_variant:Nn \fp_add:Nn  { c }`
16581 `\cs_generate_variant:Nn \fp_gadd:Nn { c }`
16582 `\cs_generate_variant:Nn \fp_sub:Nn  { c }`
16583 `\cs_generate_variant:Nn \fp_gsub:Nn { c }`

(End definition for `\fp_add:Nn` *and others. These functions are documented on page 182.)*

33.3 Showing values

`\fp_show:N` This shows the result of computing its argument. The input of `\__msg_show_-`
`\fp_show:c` `variable:NNNnn` must start with `>~` (or be empty).
`\fp_show:n`
16584 `\cs_new_protected:Npn \fp_show:N #1`
16585 `  {`
16586 `    \__msg_show_variable:NNNnn #1 \fp_if_exist:NTF ? { }`
16587 `      { > ~ \token_to_str:N #1 = \fp_to_tl:N #1 }`
16588 `  }`
16589 `\cs_new_protected:Npn \fp_show:n`
16590 `  { \__msg_show_wrap:Nn \fp_to_tl:n }`
16591 `\cs_generate_variant:Nn \fp_show:N { c }`

719

(*End definition for* `\fp_show:N` *and* `\fp_show:n`. *These functions are documented on page 188.*)

33.4 Some useful constants and scratch variables

`\c_one_fp`
`\c_e_fp` Some constants.

```
16592 \fp_const:Nn \c_e_fp          { 2.718 2818 2845 9045 }
16593 \fp_const:Nn \c_one_fp        { 1 }
```

(*End definition for* `\c_one_fp` *and* `\c_e_fp`. *These variables are documented on page 186.*)

`\c_pi_fp`
`\c_one_degree_fp` We simply round π to the closest multiple of 10^{-15}.

```
16594 \fp_const:Nn \c_pi_fp          { 3.141 5926 5358 9793 }
16595 \fp_const:Nn \c_one_degree_fp { 0.0 1745 3292 5199 4330 }
```

(*End definition for* `\c_pi_fp` *and* `\c_one_degree_fp`. *These variables are documented on page 187.*)

`\l_tmpa_fp`
`\l_tmpb_fp`
`\g_tmpa_fp`
`\g_tmpb_fp` Scratch variables are simply initialized there.

```
16596 \fp_new:N \l_tmpa_fp
16597 \fp_new:N \l_tmpb_fp
16598 \fp_new:N \g_tmpa_fp
16599 \fp_new:N \g_tmpb_fp
```

(*End definition for* `\l_tmpa_fp` *and others. These variables are documented on page 187.*)

```
16600 ⟨/initex | package⟩
```

34 l3candidates Implementation

```
16601 ⟨*initex | package⟩
```

34.1 Additions to l3basics

```
16602 ⟨@@=cs⟩
```

`\cs_log:N`
`\cs_log:c` Use `\cs_show:N` or `\cs_show:c` after calling `\__msg_log_next:` to redirect their output to the log file only. Note that `\cs_log:c` is not just a variant of `\cs_log:N` as the csname should be turned to a control sequence within a group (see `\cs_show:c`).

```
16603 \cs_new_protected:Npn \cs_log:N
16604   { \__msg_log_next: \cs_show:N }
16605 \cs_new_protected:Npn \cs_log:c
16606   { \__msg_log_next: \cs_show:c }
```

(*End definition for* `\cs_log:N`. *This function is documented on page 198.*)

`\__kernel_register_log:N`
`\__kernel_register_log:c` Redirect the output of `\__kernel_register_show:N` to the log.

```
16607 \cs_new_protected:Npn \__kernel_register_log:N
16608   { \__msg_log_next: \__kernel_register_show:N }
16609 \cs_generate_variant:Nn \__kernel_register_log:N { c }
```

(*End definition for* `\__kernel_register_log:N`.)

34.2 Additions to l3box

34.3 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.

```
16611 \fp_new:N \l__box_angle_fp
```

(*End definition for* \l__box_angle_fp.)

\l__box_cos_fp   These are used to hold the calculated sine and cosine values while carrying out a rotation.
\l__box_sin_fp

```
16612 \fp_new:N \l__box_cos_fp
16613 \fp_new:N \l__box_sin_fp
```

(*End definition for* \l__box_cos_fp *and* \l__box_sin_fp.)

\l__box_top_dim   These are the positions of the four edges of a box before manipulation.
\l__box_bottom_dim
\l__box_left_dim
\l__box_right_dim

```
16614 \dim_new:N \l__box_top_dim
16615 \dim_new:N \l__box_bottom_dim
16616 \dim_new:N \l__box_left_dim
16617 \dim_new:N \l__box_right_dim
```

(*End definition for* \l__box_top_dim *and others.*)

\l__box_top_new_dim   These are the positions of the four edges of a box after manipulation.
\l__box_bottom_new_dim
\l__box_left_new_dim
\l__box_right_new_dim

```
16618 \dim_new:N \l__box_top_new_dim
16619 \dim_new:N \l__box_bottom_new_dim
16620 \dim_new:N \l__box_left_new_dim
16621 \dim_new:N \l__box_right_new_dim
```

(*End definition for* \l__box_top_new_dim *and others.*)

\l__box_internal_box   Scratch space, but also needed by some parts of the driver.

```
16622 \box_new:N \l__box_internal_box
```

(*End definition for* \l__box_internal_box.)

\box_rotate:Nn   Rotation of a box starts with working out the relevant sine and cosine. The actual
__box_rotate:N   rotation is in an auxiliary to keep the flow slightly clearer
__box_rotate_x:nnN
__box_rotate_y:nnN
__box_rotate_quadrant_one:
__box_rotate_quadrant_two:
__box_rotate_quadrant_three:
__box_rotate_quadrant_four:

```
16623 \cs_new_protected:Npn \box_rotate:Nn #1#2
16624   {
16625     \hbox_set:Nn #1
16626       {
16627         \group_begin:
16628           \fp_set:Nn \l__box_angle_fp {#2}
16629           \fp_set:Nn \l__box_sin_fp { sind ( \l__box_angle_fp ) }
16630           \fp_set:Nn \l__box_cos_fp { cosd ( \l__box_angle_fp ) }
16631           \__box_rotate:N #1
16632         \group_end:
16633       }
16634   }
```
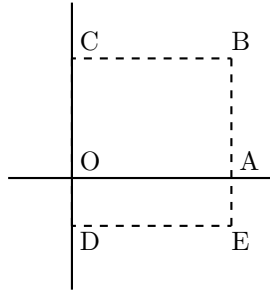
Figure 1: Co-ordinates of a box prior to rotation.

The edges of the box are then recorded: the left edge will always be at zero. Rotation of the four edges then takes place: this is most efficiently done on a quadrant by quadrant basis.

```
16635 \cs_new_protected:Npn \__box_rotate:N #1
16636   {
16637     \dim_set:Nn \l__box_top_dim    { \box_ht:N #1 }
16638     \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
16639     \dim_set:Nn \l__box_right_dim  { \box_wd:N #1 }
16640     \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:

$$P'_x = P_x - O_x$$
$$P'_y = P_y - O_y$$
$$P''_x = (P'_x \cos(\alpha)) - (P'_y \sin(\alpha))$$
$$P''_y = (P'_x \sin(\alpha)) + (P'_y \cos(\alpha))$$
$$P'''_x = P''_x + O_x + L_x$$
$$P'''_y = P''_y + O_y$$

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.)

```
16641       \fp_compare:nNnTF \l__box_sin_fp > \c_zero_fp
16642         {
16643           \fp_compare:nNnTF \l__box_cos_fp > \c_zero_fp
16644             { \__box_rotate_quadrant_one: }
16645             { \__box_rotate_quadrant_two: }
16646         }
16647         {
16648           \fp_compare:nNnTF \l__box_cos_fp < \c_zero_fp
16649             { \__box_rotate_quadrant_three: }
16650             { \__box_rotate_quadrant_four: }
16651         }
```

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 will be in the same place as the original.

```
16652        \hbox_set:Nn \l__box_internal_box { \box_use:N #1 }
16653        \hbox_set:Nn \l__box_internal_box
16654          {
16655            \tex_kern:D -\l__box_left_new_dim
16656            \hbox:n
16657              {
16658                \__driver_box_use_rotate:Nn
16659                  \l__box_internal_box
16660                  \l__box_angle_fp
16661              }
16662          }
```

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.

```
16663        \box_set_ht:Nn \l__box_internal_box {  \l__box_top_new_dim }
16664        \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
16665        \box_set_wd:Nn \l__box_internal_box
16666          { \l__box_right_new_dim - \l__box_left_new_dim }
16667        \box_use:N \l__box_internal_box
16668      }
```

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.

```
16669 \cs_new_protected:Npn \__box_rotate_x:nnN #1#2#3
16670   {
16671      \dim_set:Nn #3
16672        {
16673          \fp_to_dim:n
16674            {
16675                \l__box_cos_fp * \dim_to_fp:n {#1}
16676              - \l__box_sin_fp * \dim_to_fp:n {#2}
16677            }
16678        }
16679   }
16680 \cs_new_protected:Npn \__box_rotate_y:nnN #1#2#3
16681   {
16682      \dim_set:Nn #3
16683        {
16684          \fp_to_dim:n
16685            {
16686                \l__box_sin_fp * \dim_to_fp:n {#1}
16687              + \l__box_cos_fp * \dim_to_fp:n {#2}
16688            }
16689        }
16690   }
```

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.

```
16691 \cs_new_protected:Npn \__box_rotate_quadrant_one:
16692   {
16693     \__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
16694       \l__box_top_new_dim
16695     \__box_rotate_y:nnN \l__box_left_dim  \l__box_bottom_dim
16696       \l__box_bottom_new_dim
16697     \__box_rotate_x:nnN \l__box_left_dim  \l__box_top_dim
16698       \l__box_left_new_dim
16699     \__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
16700       \l__box_right_new_dim
16701   }
16702 \cs_new_protected:Npn \__box_rotate_quadrant_two:
16703   {
16704     \__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
16705       \l__box_top_new_dim
16706     \__box_rotate_y:nnN \l__box_left_dim  \l__box_top_dim
16707       \l__box_bottom_new_dim
16708     \__box_rotate_x:nnN \l__box_right_dim  \l__box_top_dim
16709       \l__box_left_new_dim
16710     \__box_rotate_x:nnN \l__box_left_dim    \l__box_bottom_dim
16711       \l__box_right_new_dim
16712   }
16713 \cs_new_protected:Npn \__box_rotate_quadrant_three:
16714   {
16715     \__box_rotate_y:nnN \l__box_left_dim  \l__box_bottom_dim
16716       \l__box_top_new_dim
16717     \__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
16718       \l__box_bottom_new_dim
16719     \__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
16720       \l__box_left_new_dim
16721     \__box_rotate_x:nnN \l__box_left_dim    \l__box_top_dim
16722       \l__box_right_new_dim
16723   }
16724 \cs_new_protected:Npn \__box_rotate_quadrant_four:
16725   {
16726     \__box_rotate_y:nnN \l__box_left_dim  \l__box_top_dim
16727       \l__box_top_new_dim
16728     \__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
16729       \l__box_bottom_new_dim
16730     \__box_rotate_x:nnN \l__box_left_dim  \l__box_bottom_dim
16731       \l__box_left_new_dim
16732     \__box_rotate_x:nnN \l__box_right_dim \l__box_top_dim
16733       \l__box_right_new_dim
16734   }
```

(*End definition for* \box_rotate:Nn *and others. These functions are documented on page 200.*)

\l__box_scale_x_fp Scaling is potentially-different in the two axes.
\l__box_scale_y_fp
```
16735 \fp_new:N \l__box_scale_x_fp
16736 \fp_new:N \l__box_scale_y_fp
```

(*End definition for* \l__box_scale_x_fp *and* \l__box_scale_y_fp.)

Resizing a box starts by working out the various dimensions of the existing box.

```
16737 \cs_new_protected:Npn \box_resize:Nnn #1#2#3
16738   {
16739     \hbox_set:Nn #1
16740       {
16741         \group_begin:
16742           \__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.

```
16743         \fp_set:Nn \l__box_scale_x_fp
16744           { \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.

```
16745         \fp_set:Nn \l__box_scale_y_fp
16746           {
16747             \dim_to_fp:n {#3}
16748             / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
16749           }
```

Hand off to the auxiliary which does the rest of the work.

```
16750         \__box_resize:N #1
16751       \group_end:
16752       }
16753   }
16754 \cs_generate_variant:Nn \box_resize:Nnn { c }
16755 \cs_new_protected:Npn \__box_resize_set_corners:N #1
16756   {
16757     \dim_set:Nn \l__box_top_dim    { \box_ht:N #1 }
16758     \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
16759     \dim_set:Nn \l__box_right_dim  { \box_wd:N #1 }
16760     \dim_zero:N \l__box_left_dim
16761   }
```

With at least one real scaling to do, the next phase is to find the new edge co-ordinates. 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.

```
16762 \cs_new_protected:Npn \__box_resize:N #1
16763   {
16764     \__box_resize:NNN \l__box_right_new_dim
16765       \l__box_scale_x_fp \l__box_right_dim
16766     \__box_resize:NNN \l__box_bottom_new_dim
16767       \l__box_scale_y_fp \l__box_bottom_dim
16768     \__box_resize:NNN \l__box_top_new_dim
16769       \l__box_scale_y_fp \l__box_top_dim
16770     \__box_resize_common:N #1
16771   }
16772 \cs_new_protected:Npn \__box_resize:NNN #1#2#3
16773   {
16774     \dim_set:Nn #1
16775       { \fp_to_dim:n { \fp_abs:n { #2 } * \dim_to_fp:n { #3 } } }
16776   }
```

(*End definition for* \box_resize:Nnn *and others. These functions are documented on page 199.*)

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).

```
16777 \cs_new_protected:Npn \box_resize_to_ht:Nn #1#2
16778   {
16779     \hbox_set:Nn #1
16780       {
16781         \group_begin:
16782           \__box_resize_set_corners:N #1
16783           \fp_set:Nn \l__box_scale_y_fp
16784             {
16785                 \dim_to_fp:n {#2}
16786               / \dim_to_fp:n { \l__box_top_dim }
16787             }
16788           \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
16789           \__box_resize:N #1
16790         \group_end:
16791       }
16792   }
16793 \cs_generate_variant:Nn \box_resize_to_ht:Nn { c }
16794 \cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2
16795   {
16796     \hbox_set:Nn #1
16797       {
16798         \group_begin:
16799           \__box_resize_set_corners:N #1
16800           \fp_set:Nn \l__box_scale_y_fp
16801             {
16802                 \dim_to_fp:n {#2}
16803               / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
16804             }
16805           \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
16806           \__box_resize:N #1
16807         \group_end:
16808       }
16809   }
16810 \cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
16811 \cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
16812   {
16813     \hbox_set:Nn #1
16814       {
16815         \group_begin:
16816           \__box_resize_set_corners:N #1
16817           \fp_set:Nn \l__box_scale_x_fp
16818             { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
16819           \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
16820           \__box_resize:N #1
16821         \group_end:
16822       }
16823   }
16824 \cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
16825 \cs_new_protected:Npn \box_resize_to_wd_and_ht:Nnn #1#2#3
16826   {
```

726

```
16827        \hbox_set:Nn #1
16828          {
16829            \group_begin:
16830              \__box_resize_set_corners:N #1
16831              \fp_set:Nn \l__box_scale_x_fp
16832                { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
16833              \fp_set:Nn \l__box_scale_y_fp
16834                {
16835                    \dim_to_fp:n {#3}
16836                  / \dim_to_fp:n { \l__box_top_dim }
16837                }
16838              \__box_resize:N #1
16839            \group_end:
16840          }
16841      }
16842  \cs_generate_variant:Nn \box_resize_to_wd_and_ht:Nnn { c }
```

(*End definition for* `\box_resize_to_ht:Nn` *and others. These functions are documented on page 199.*)

`\box_scale:Nnn`
`\box_scale:cnn`
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 dimension scaling operations are carried out using the TeX mechanism as it avoids needing to use too many `fp` operations.

```
16843  \cs_new_protected:Npn \box_scale:Nnn #1#2#3
16844    {
16845      \hbox_set:Nn #1
16846        {
16847          \group_begin:
16848            \fp_set:Nn \l__box_scale_x_fp {#2}
16849            \fp_set:Nn \l__box_scale_y_fp {#3}
16850            \dim_set:Nn \l__box_top_dim    {  \box_ht:N #1 }
16851            \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
16852            \dim_set:Nn \l__box_right_dim  {  \box_wd:N #1 }
16853            \dim_zero:N \l__box_left_dim
16854            \dim_set:Nn \l__box_top_new_dim
16855              { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
16856            \dim_set:Nn \l__box_bottom_new_dim
16857              { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
16858            \dim_set:Nn \l__box_right_new_dim
16859                { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
16860            \__box_resize_common:N #1
16861          \group_end:
16862        }
16863    }
16864  \cs_generate_variant:Nn \box_scale:Nnn { c }
```

(*End definition for* `\box_scale:Nnn`. *This function is documented on page 200.*)

`\__box_resize_common:N` The main resize function places in input into a box which will start of with zero width, and includes the handles for engine rescaling.

```
16865  \cs_new_protected:Npn \__box_resize_common:N #1
16866    {
16867      \hbox_set:Nn \l__box_internal_box
```

727

```
16868            {
16869              \__driver_box_use_scale:Nnn
16870                #1
16871                \l__box_scale_x_fp
16872                \l__box_scale_y_fp
16873            }
```
The new height and depth can be applied directly.
```
16874            \fp_compare:nNnTF \l__box_scale_y_fp > \c_zero_fp
16875              {
16876                \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
16877                \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
16878              }
16879              {
16880                \box_set_dp:Nn \l__box_internal_box { \l__box_top_new_dim }
16881                \box_set_ht:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
16882              }
```
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.
```
16883            \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
16884              {
16885                \hbox_to_wd:nn { \l__box_right_new_dim }
16886                  {
16887                    \tex_kern:D \l__box_right_new_dim
16888                    \box_use:N \l__box_internal_box
16889                    \tex_hss:D
16890                  }
16891              }
16892              {
16893                \box_set_wd:Nn \l__box_internal_box { \l__box_right_new_dim }
16894                \hbox:n
16895                  {
16896                    \tex_kern:D \c_zero_dim
16897                    \box_use:N \l__box_internal_box
16898                    \tex_hss:D
16899                  }
16900              }
16901          }
```
(End definition for __box_resize_common:N.*)*

34.4   Viewing part of a box

\box_clip:N   A wrapper around the driver-dependent code.

\box_clip:c
```
16902 \cs_new_protected:Npn \box_clip:N #1
16903   { \hbox_set:Nn #1 { \__driver_box_use_clip:N #1 } }
16904 \cs_generate_variant:Nn \box_clip:N { c }
```
(End definition for \box_clip:N. *This function is documented on page 200.)*

`\box_trim:Nnnnn`
`\box_trim:cnnnn`

Trimming from the left- and right-hand edges of the box is easy: kern the appropriate parts off each side.

```
16905 \cs_new_protected:Npn \box_trim:Nnnnn #1#2#3#4#5
16906   {
16907     \hbox_set:Nn \l__box_internal_box
16908       {
16909         \tex_kern:D -\__dim_eval:w #2 \__dim_eval_end:
16910         \box_use:N #1
16911         \tex_kern:D -\__dim_eval:w #4 \__dim_eval_end:
16912       }
```

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`.

```
16913     \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
16914       {
16915         \hbox_set:Nn \l__box_internal_box
16916           {
16917             \box_move_down:nn \c_zero_dim
16918               { \box_use:N \l__box_internal_box }
16919           }
16920         \box_set_dp:Nn \l__box_internal_box { \box_dp:N #1 - (#3) }
16921       }
16922       {
16923         \hbox_set:Nn \l__box_internal_box
16924           {
16925             \box_move_down:nn { #3 - \box_dp:N #1 }
16926               { \box_use:N \l__box_internal_box }
16927           }
16928         \box_set_dp:Nn \l__box_internal_box \c_zero_dim
16929       }
```

Same thing, this time from the top of the box.

```
16930     \dim_compare:nNnTF { \box_ht:N \l__box_internal_box } > {#5}
16931       {
16932         \hbox_set:Nn \l__box_internal_box
16933           {
16934             \box_move_up:nn \c_zero_dim
16935               { \box_use:N \l__box_internal_box }
16936           }
16937         \box_set_ht:Nn \l__box_internal_box
16938           { \box_ht:N \l__box_internal_box - (#5) }
16939       }
16940       {
16941         \hbox_set:Nn \l__box_internal_box
16942           {
16943             \box_move_up:nn { #5 - \box_ht:N \l__box_internal_box }
16944               { \box_use:N \l__box_internal_box }
16945           }
16946         \box_set_ht:Nn \l__box_internal_box \c_zero_dim
16947       }
```

```
16948          \box_set_eq:NN #1 \l__box_internal_box
16949        }
16950  \cs_generate_variant:Nn \box_trim:Nnnnn { c }
```

(*End definition for* \box_trim:Nnnnn. *This function is documented on page 200.*)

\box_viewport:Nnnnn   The same general logic as for the trim operation, but with absolute dimensions. As a
\box_viewport:cnnnn   result, there are some things to watch out for in the vertical direction.

```
16951  \cs_new_protected:Npn \box_viewport:Nnnnn #1#2#3#4#5
16952    {
16953      \hbox_set:Nn \l__box_internal_box
16954        {
16955          \tex_kern:D -\__dim_eval:w #2 \__dim_eval_end:
16956          \box_use:N #1
16957          \tex_kern:D \__dim_eval:w #4 - \box_wd:N #1 \__dim_eval_end:
16958        }
16959      \dim_compare:nNnTF {#3} < \c_zero_dim
16960        {
16961          \hbox_set:Nn \l__box_internal_box
16962            {
16963              \box_move_down:nn \c_zero_dim
16964                { \box_use:N \l__box_internal_box }
16965            }
16966          \box_set_dp:Nn \l__box_internal_box { -\dim_eval:n {#3} }
16967        }
16968        {
16969          \hbox_set:Nn \l__box_internal_box
16970            { \box_move_down:nn {#3} { \box_use:N \l__box_internal_box } }
16971          \box_set_dp:Nn \l__box_internal_box \c_zero_dim
16972        }
16973      \dim_compare:nNnTF {#5} > \c_zero_dim
16974        {
16975          \hbox_set:Nn \l__box_internal_box
16976            {
16977              \box_move_up:nn \c_zero_dim
16978                { \box_use:N \l__box_internal_box }
16979            }
16980          \box_set_ht:Nn \l__box_internal_box
16981            {
16982              #5
16983              \dim_compare:nNnT {#3} > \c_zero_dim
16984                { - (#3) }
16985            }
16986        }
16987        {
16988          \hbox_set:Nn \l__box_internal_box
16989            {
16990              \box_move_up:nn { -\dim_eval:n {#5} }
16991                { \box_use:N \l__box_internal_box }
16992            }
16993          \box_set_ht:Nn \l__box_internal_box \c_zero_dim
16994        }
16995      \box_set_eq:NN #1 \l__box_internal_box
16996    }
16997  \cs_generate_variant:Nn \box_viewport:Nnnnn { c }
```

730

(*End definition for* `\box_viewport:Nnnnn`. *This function is documented on page 201.*)

34.5 Additions to **l3clist**

`16998 ⟨@@=clist⟩`

`\clist_log:N` Redirect output of `\clist_show:N` to the log.
`\clist_log:c`
`\clist_log:n`
```
16999 \cs_new_protected:Npn \clist_log:N
17000   { \__msg_log_next: \clist_show:N }
17001 \cs_new_protected:Npn \clist_log:n
17002   { \__msg_log_next: \clist_show:n }
17003 \cs_generate_variant:Nn \clist_log:N { c }
```

(*End definition for* `\clist_log:N` *and* `\clist_log:n`. *These functions are documented on page 201.*)

34.6 Additions to **l3coffins**

`17004 ⟨@@=coffin⟩`

34.7 Rotating coffins

`\l__coffin_sin_fp` Used for rotations to get the sine and cosine values.
`\l__coffin_cos_fp`
```
17005 \fp_new:N \l__coffin_sin_fp
17006 \fp_new:N \l__coffin_cos_fp
```

(*End 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.

`17007 \prop_new:N \l__coffin_bounding_prop`

(*End definition for* `\l__coffin_bounding_prop`.)

`\l__coffin_bounding_shift_dim` The shift of the bounding box of a coffin from the real content.

`17008 \dim_new:N \l__coffin_bounding_shift_dim`

(*End definition for* `\l__coffin_bounding_shift_dim`.)

`\l__coffin_left_corner_dim` These are used to hold maxima for the various corner values: these thus define the
`\l__coffin_right_corner_dim` minimum size of the bounding box after rotation.
`\l__coffin_bottom_corner_dim`
`\l__coffin_top_corner_dim`
```
17009 \dim_new:N \l__coffin_left_corner_dim
17010 \dim_new:N \l__coffin_right_corner_dim
17011 \dim_new:N \l__coffin_bottom_corner_dim
17012 \dim_new:N \l__coffin_top_corner_dim
```

(*End definition for* `\l__coffin_left_corner_dim` *and others.*)

`\coffin_rotate:Nn` Rotating a coffin requires several steps which can be conveniently run together. The sine
`\coffin_rotate:cn` 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.
```
17013 \cs_new_protected:Npn \coffin_rotate:Nn #1#2
17014   {
17015     \fp_set:Nn \l__coffin_sin_fp { sind ( #2 ) }
17016     \fp_set:Nn \l__coffin_cos_fp { cosd ( #2 ) }
```

731

The corners and poles of the coffin can now be rotated around the origin. This is best achieved using mapping functions.

```
17017       \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
17018         { \__coffin_rotate_corner:Nnnn #1 {##1} ##2 }
17019       \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
17020         { \__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.

```
17021       \__coffin_set_bounding:N #1
17022       \prop_map_inline:Nn \l__coffin_bounding_prop
17023         { \__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.

```
17024       \__coffin_find_corner_maxima:N #1
17025       \__coffin_find_bounding_shift:
17026       \box_rotate:Nn #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.

```
17027       \hbox_set:Nn \l__coffin_internal_box
17028         {
17029           \tex_kern:D
17030             \__dim_eval:w
17031               \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim
17032             \__dim_eval_end:
17033           \box_move_down:nn { \l__coffin_bottom_corner_dim }
17034             { \box_use:N #1 }
17035         }
```

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.

```
17036       \box_set_ht:Nn \l__coffin_internal_box
17037         { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
17038       \box_set_dp:Nn \l__coffin_internal_box { 0 pt }
17039       \box_set_wd:Nn \l__coffin_internal_box
17040         { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
17041       \hbox_set:Nn #1 { \box_use: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.

```
17042       \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
17043         { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
17044       \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
17045         { \__coffin_shift_pole:Nnnnnn #1 {##1} ##2 }
```

```
17046        }
17047 \cs_generate_variant:Nn \coffin_rotate:Nn { c }
```

(*End definition for* \coffin_rotate:Nn. *This function is documented on page 201.*)

__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.

```
17048 \cs_new_protected:Npn \__coffin_set_bounding:N #1
17049   {
17050        \prop_put:Nnx \l__coffin_bounding_prop { tl }
17051          { { 0 pt } { \dim_eval:n { \box_ht:N #1 } } }
17052        \prop_put:Nnx \l__coffin_bounding_prop { tr }
17053          { { \dim_eval:n { \box_wd:N #1 } } { \dim_eval:n { \box_ht:N #1 } } }
17054        \dim_set:Nn \l__coffin_internal_dim { -\box_dp:N #1 }
17055        \prop_put:Nnx \l__coffin_bounding_prop { bl }
17056          { { 0 pt } { \dim_use:N \l__coffin_internal_dim } }
17057        \prop_put:Nnx \l__coffin_bounding_prop { br }
17058          { { \dim_eval:n { \box_wd:N #1 } } { \dim_use:N \l__coffin_internal_dim } }
17059   }
```

(*End definition for* __coffin_set_bounding:N.)

__coffin_rotate_bounding:nnn   Rotating the position of the corner of the coffin is just a case of treating this as a vector
__coffin_rotate_corner:Nnnn   from the reference point. The same treatment is used for the corners of the material itself
and the bounding box.

```
17060 \cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
17061   {
17062        \__coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
17063        \prop_put:Nnx \l__coffin_bounding_prop {#1}
17064          { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
17065   }
17066 \cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
17067   {
17068        \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
17069        \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
17070          { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
17071   }
```

(*End definition for* __coffin_rotate_bounding:nnn *and* __coffin_rotate_corner:Nnnn.)

__coffin_rotate_pole:Nnnnnn   Rotating a single pole simply means shifting the co-ordinate of the pole and its direction.
The rotation here is about the bottom-left corner of the coffin.

```
17072 \cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
17073   {
17074        \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
17075        \__coffin_rotate_vector:nnNN {#5} {#6}
17076          \l__coffin_x_prime_dim \l__coffin_y_prime_dim
17077        \__coffin_set_pole:Nnx #1 {#2}
17078          {
17079            { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
17080            { \dim_use:N \l__coffin_x_prime_dim }
17081            { \dim_use:N \l__coffin_y_prime_dim }
17082          }
17083   }
```

(End 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.

```
17084 \cs_new_protected:Npn \__coffin_rotate_vector:nnNN #1#2#3#4
17085   {
17086     \dim_set:Nn #3
17087       {
17088         \fp_to_dim:n
17089           {
17090               \dim_to_fp:n {#1} * \l__coffin_cos_fp
17091             - \dim_to_fp:n {#2} * \l__coffin_sin_fp
17092           }
17093       }
17094     \dim_set:Nn #4
17095       {
17096         \fp_to_dim:n
17097           {
17098               \dim_to_fp:n {#1} * \l__coffin_sin_fp
17099             + \dim_to_fp:n {#2} * \l__coffin_cos_fp
17100           }
17101       }
17102   }
```

(End definition for `\__coffin_rotate_vector:nnNN`.*)*

`\__coffin_find_corner_maxima:N` The idea here is to find the extremities of the content of the coffin. This is done by
`\__coffin_find_corner_maxima_aux:nn` 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.

```
17103 \cs_new_protected:Npn \__coffin_find_corner_maxima:N #1
17104   {
17105     \dim_set:Nn \l__coffin_top_corner_dim    { -\c_max_dim }
17106     \dim_set:Nn \l__coffin_right_corner_dim  { -\c_max_dim }
17107     \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
17108     \dim_set:Nn \l__coffin_left_corner_dim   { \c_max_dim }
17109     \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
17110       { \__coffin_find_corner_maxima_aux:nn ##2 }
17111   }
17112 \cs_new_protected:Npn \__coffin_find_corner_maxima_aux:nn #1#2
17113   {
17114     \dim_set:Nn \l__coffin_left_corner_dim
17115       { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
17116     \dim_set:Nn \l__coffin_right_corner_dim
17117       { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
17118     \dim_set:Nn \l__coffin_bottom_corner_dim
17119       { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
17120     \dim_set:Nn \l__coffin_top_corner_dim
17121       { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
17122   }
```

(End definition for `\__coffin_find_corner_maxima:N` *and* `\__coffin_find_corner_maxima_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.

```
17123 \cs_new_protected:Npn \__coffin_find_bounding_shift:
17124   {
17125     \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }
17126     \prop_map_inline:Nn \l__coffin_bounding_prop
17127       { \__coffin_find_bounding_shift_aux:nn ##2 }
17128   }
17129 \cs_new_protected:Npn \__coffin_find_bounding_shift_aux:nn #1#2
17130   {
17131     \dim_set:Nn \l__coffin_bounding_shift_dim
17132       { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
17133   }
```

(End definition for __coffin_find_bounding_shift: *and* __coffin_find_bounding_shift_aux:nn*.)*

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.

```
17134 \cs_new_protected:Npn \__coffin_shift_corner:Nnnn #1#2#3#4
17135   {
17136     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _ prop } {#2}
17137       {
17138         { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
17139         { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
17140       }
17141   }
17142 \cs_new_protected:Npn \__coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
17143   {
17144     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _ prop } {#2}
17145       {
17146         { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
17147         { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
17148         {#5} {#6}
17149       }
17150   }
```

(End definition for __coffin_shift_corner:Nnnn *and* __coffin_shift_pole:Nnnnnn*.)*

34.8 Resizing coffins

Storage for the scaling factors in x and y, respectively.

```
17151 \fp_new:N \l__coffin_scale_x_fp
17152 \fp_new:N \l__coffin_scale_y_fp
```

(End definition for \l__coffin_scale_x_fp *and* \l__coffin_scale_y_fp*.)*

When scaling, the values given have to be turned into absolute values.

```
17153 \dim_new:N \l__coffin_scaled_total_height_dim
17154 \dim_new:N \l__coffin_scaled_width_dim
```

(End definition for \l__coffin_scaled_total_height_dim *and* \l__coffin_scaled_width_dim*.)*

\coffin_resize:Nnn
\coffin_resize:cnn
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.

```
17155 \cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
17156   {
17157     \fp_set:Nn \l__coffin_scale_x_fp
17158       { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
17159     \fp_set:Nn \l__coffin_scale_y_fp
17160       {
17161           \dim_to_fp:n {#3}
17162         / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
17163       }
17164     \box_resize:Nnn #1 {#2} {#3}
17165     \__coffin_resize_common:Nnn #1 {#2} {#3}
17166   }
17167 \cs_generate_variant:Nn \coffin_resize:Nnn { c }
```

(*End definition for* \coffin_resize:Nnn. *This function is documented on page 201.*)

__coffin_resize_common:Nnn
The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```
17168 \cs_new_protected:Npn \__coffin_resize_common:Nnn #1#2#3
17169   {
17170     \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
17171       { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }
17172     \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
17173       { \__coffin_scale_pole:Nnnnnn #1 {##1} ##2 }
```

Negative *x*-scaling values will place the poles in the wrong location: this is corrected here.

```
17174     \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
17175       {
17176         \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
17177           { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
17178         \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
17179           { \__coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
17180       }
17181   }
```

(*End definition for* __coffin_resize_common:Nnn.)

\coffin_scale:Nnn
\coffin_scale:cnn
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.

```
17182 \cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
17183   {
17184     \fp_set:Nn \l__coffin_scale_x_fp {#2}
17185     \fp_set:Nn \l__coffin_scale_y_fp {#3}
17186     \box_scale:Nnn #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }
17187     \dim_set:Nn \l__coffin_internal_dim
17188       { \coffin_ht:N #1 + \coffin_dp:N #1 }
17189     \dim_set:Nn \l__coffin_scaled_total_height_dim
```

```
17190            { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_internal_dim }
17191        \dim_set:Nn \l__coffin_scaled_width_dim
17192            { -\fp_abs:n { \l__coffin_scale_x_fp  } \coffin_wd:N #1 }
17193        \__coffin_resize_common:Nnn #1
17194            { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
17195      }
17196 \cs_generate_variant:Nn \coffin_scale:Nnn { c }
```

(*End definition for* `\coffin_scale:Nnn`. *This function is documented on page 201.*)

`\__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.

```
17197 \cs_new_protected:Npn \__coffin_scale_vector:nnNN #1#2#3#4
17198   {
17199     \dim_set:Nn #3
17200        { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
17201     \dim_set:Nn #4
17202        { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
17203   }
```

(*End definition for* `\__coffin_scale_vector:nnNN`.)

`\__coffin_scale_corner:Nnnn` Scaling both corners and poles is a simple calculation using the preceding vector scaling.
`\__coffin_scale_pole:Nnnnnn`
```
17204 \cs_new_protected:Npn \__coffin_scale_corner:Nnnn #1#2#3#4
17205   {
17206     \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
17207     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
17208        { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
17209   }
17210 \cs_new_protected:Npn \__coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
17211   {
17212     \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
17213     \__coffin_set_pole:Nnx #1 {#2}
17214        {
17215          { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
17216          {#5} {#6}
17217        }
17218   }
```

(*End definition for* `\__coffin_scale_corner:Nnnn` *and* `\__coffin_scale_pole:Nnnnnn`.)

`\__coffin_x_shift_corner:Nnnn` These functions correct for the x displacement that takes place with a negative horizontal
`\__coffin_x_shift_pole:Nnnnnn` scaling.

```
17219 \cs_new_protected:Npn \__coffin_x_shift_corner:Nnnn #1#2#3#4
17220   {
17221     \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
17222        {
17223          { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
17224        }
17225   }
17226 \cs_new_protected:Npn \__coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
17227   {
17228     \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } {#2}
```

```
17229        {
17230          { \dim_eval:n #3 + \box_wd:N #1 } {#4}
17231          {#5} {#6}
17232        }
17233    }
```

(*End definition for* __coffin_x_shift_corner:Nnnn *and* __coffin_x_shift_pole:Nnnnnn.)

34.9 Coffin diagnostics

\coffin_log_structure:N   Redirect output of \coffin_show_structure:N to the log.
\coffin_log_structure:c
```
17234 \cs_new_protected:Npn \coffin_log_structure:N
17235    { \__msg_log_next: \coffin_show_structure:N }
17236 \cs_generate_variant:Nn \coffin_log_structure:N { c }
```

(*End definition for* \coffin_log_structure:N. *This function is documented on page 201.*)

34.10 Additions to l3file

```
17237 ⟨@@=file⟩
```

\file_if_exist_input:nTF   Input of a file with a test for existence cannot be done the usual way as the tokens to
insert are in an odd place.
```
17238 \cs_new_protected:Npn \file_if_exist_input:n #1
17239    {
17240      \file_if_exist:nT {#1}
17241        { \__file_input:V \l__file_internal_name_tl }
17242    }
17243 \cs_new_protected:Npn \file_if_exist_input:nT #1#2
17244    {
17245      \file_if_exist:nT {#1}
17246        {
17247          #2
17248          \__file_input:V \l__file_internal_name_tl
17249        }
17250    }
17251 \cs_new_protected:Npn \file_if_exist_input:nF #1
17252    {
17253      \file_if_exist:nTF {#1}
17254        { \__file_input:V \l__file_internal_name_tl }
17255    }
17256 \cs_new_protected:Npn \file_if_exist_input:nTF #1#2
17257    {
17258      \file_if_exist:nTF {#1}
17259        {
17260          #2
17261          \__file_input:V \l__file_internal_name_tl
17262        }
17263    }
```

(*End definition for* \file_if_exist_input:nTF. *This function is documented on page 202.*)

```
17264 ⟨@@=ior⟩
```

\ior_map_break:
\ior_map_break:n

Usual map breaking functions. Those are not yet in l3kernel proper since the mapping below is the first of its kind.

```
17265 \cs_new:Npn \ior_map_break:
17266   { \__prg_map_break:Nn \ior_map_break: { } }
17267 \cs_new:Npn \ior_map_break:n
17268   { \__prg_map_break:Nn \ior_map_break: }
```

(*End definition for* \ior_map_break: *and* \ior_map_break:n. *These functions are documented on page* *202.*)

\ior_map_inline:Nn
\ior_str_map_inline:Nn
__ior_map_inline:NNn
__ior_map_inline:NNNn
__ior_map_inline_loop:NNN
\l__ior_internal_tl

Mapping to 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. This mapping cannot be nested as the stream has only one "current line".

```
17269 \cs_new_protected:Npn \ior_map_inline:Nn
17270   { \__ior_map_inline:NNn \ior_get:NN }
17271 \cs_new_protected:Npn \ior_str_map_inline:Nn
17272   { \__ior_map_inline:NNn \ior_get_str:NN }
17273 \cs_new_protected:Npn \__ior_map_inline:NNn
17274   {
17275     \int_gincr:N \g__prg_map_int
17276     \exp_args:Nc \__ior_map_inline:NNNn
17277       { __prg_map_ \int_use:N \g__prg_map_int :n }
17278   }
17279 \cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4
17280   {
17281     \cs_set:Npn #1 ##1 {#4}
17282     \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
17283     \__prg_break_point:Nn \ior_map_break:
17284       { \int_gdecr:N \g__prg_map_int }
17285   }
17286 \cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
17287   {
17288     #2 #3 \l__ior_internal_tl
17289     \ior_if_eof:NF #3
17290       {
17291         \exp_args:No #1 \l__ior_internal_tl
17292         \__ior_map_inline_loop:NNN #1#2#3
17293       }
17294   }
17295 \tl_new:N  \l__ior_internal_tl
```

(*End definition for* \ior_map_inline:Nn *and others. These functions are documented on page* *202.*)

\ior_log_streams: Redirect output of \ior_list_streams: to the log.

```
17296 \cs_new_protected:Npn \ior_log_streams:
17297   { \__msg_log_next: \ior_list_streams: }
```

(*End definition for* \ior_log_streams:. *This function is documented on page* *203.*)

```
17298 ⟨@@=iow⟩
```

\iow_log_streams: Redirect output of \iow_list_streams: to the log.

```
17299 \cs_new_protected:Npn \iow_log_streams:
17300   { \__msg_log_next: \iow_list_streams: }
```

(*End definition for* \iow_log_streams:. *This function is documented on page* *203.*)

739

\ior_map_break:
\ior_map_break:n

Usual map breaking functions. Those are not yet in l3kernel proper since the mapping below is the first of its kind.

```
17265 \cs_new:Npn \ior_map_break:
17266   { \__prg_map_break:Nn \ior_map_break: { } }
17267 \cs_new:Npn \ior_map_break:n
17268   { \__prg_map_break:Nn \ior_map_break: }
```

(*End definition for* \ior_map_break: *and* \ior_map_break:n. *These functions are documented on page 202.*)

\ior_map_inline:Nn
\ior_str_map_inline:Nn
__ior_map_inline:NNn
__ior_map_inline:NNNn
__ior_map_inline_loop:NNN
\l__ior_internal_tl

Mapping to 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. This mapping cannot be nested as the stream has only one "current line".

```
17269 \cs_new_protected:Npn \ior_map_inline:Nn
17270   { \__ior_map_inline:NNn \ior_get:NN }
17271 \cs_new_protected:Npn \ior_str_map_inline:Nn
17272   { \__ior_map_inline:NNn \ior_get_str:NN }
17273 \cs_new_protected:Npn \__ior_map_inline:NNn
17274   {
17275     \int_gincr:N \g__prg_map_int
17276     \exp_args:Nc \__ior_map_inline:NNNn
17277       { __prg_map_ \int_use:N \g__prg_map_int :n }
17278   }
17279 \cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4
17280   {
17281     \cs_set:Npn #1 ##1 {#4}
17282     \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
17283     \__prg_break_point:Nn \ior_map_break:
17284       { \int_gdecr:N \g__prg_map_int }
17285   }
17286 \cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
17287   {
17288     #2 #3 \l__ior_internal_tl
17289     \ior_if_eof:NF #3
17290       {
17291         \exp_args:No #1 \l__ior_internal_tl
17292         \__ior_map_inline_loop:NNN #1#2#3
17293       }
17294   }
17295 \tl_new:N  \l__ior_internal_tl
```

(*End definition for* \ior_map_inline:Nn *and others. These functions are documented on page 202.*)

\ior_log_streams: Redirect output of \ior_list_streams: to the log.

```
17296 \cs_new_protected:Npn \ior_log_streams:
17297   { \__msg_log_next: \ior_list_streams: }
```

(*End definition for* \ior_log_streams:. *This function is documented on page 203.*)

```
17298 ⟨@@=iow⟩
```

\iow_log_streams: Redirect output of \iow_list_streams: to the log.

```
17299 \cs_new_protected:Npn \iow_log_streams:
17300   { \__msg_log_next: \iow_list_streams: }
```

(*End definition for* \iow_log_streams:. *This function is documented on page 203.*)

34.11 Additions to **l3fp-assign**

17301 ⟨@@=fp⟩

\fp_log:N Redirect output of \fp_show:N to the log.
\fp_log:c
\fp_log:n
17302 \cs_new_protected:Npn \fp_log:N
17303   { __msg_log_next: \fp_show:N }
17304 \cs_new_protected:Npn \fp_log:n
17305   { __msg_log_next: \fp_show:n }
17306 \cs_generate_variant:Nn \fp_log:N { c }

(*End definition for* \fp_log:N *and* \fp_log:n*. These functions are documented on page 203.*)

34.12 Additions to **l3int**

\int_log:N Redirect output of \int_show:N to the log. This is not just a copy of __kernel_-
\int_log:c register_log:N because of subtleties involving \currentgrouplevel and \currentgrouptype.
See \int_show:N for details.

17307 \cs_new_protected:Npn \int_log:N
17308   { __msg_log_next: \int_show:N }
17309 \cs_generate_variant:Nn \int_log:N { c }

(*End definition for* \int_log:N*. This function is documented on page 203.*)

\int_log:n Redirect output of \int_show:n to the log.

17310 \cs_new_protected:Npn \int_log:n
17311   { __msg_log_next: \int_show:n }

(*End definition for* \int_log:n*. This function is documented on page 203.*)

34.13 Additions to **l3keys**

17312 ⟨@@=keys⟩

\keys_log:nn Redirect output of \keys_show:nn to the log.

17313 \cs_new_protected:Npn \keys_log:nn
17314   { __msg_log_next: \keys_show:nn }

(*End definition for* \keys_log:nn*. This function is documented on page 204.*)

34.14 Additions to **l3msg**

17315 ⟨@@=msg⟩

\msg_expandable_error:nnnnnn Pass to an auxiliary the message to display and the module name
\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
__msg_expandable_error_module:nn

17316 \cs_new:Npn \msg_expandable_error:nnnnnn #1#2#3#4#5#6
17317   {
17318   \exp_args:Nf __msg_expandable_error_module:nn
17319   {
17320   \exp_args:Nf \tl_to_str:n
17321   { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
17322   }
17323   {#1}
17324   }
17325 \cs_new:Npn \msg_expandable_error:nnnnn #1#2#3#4#5
17326   { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }

740

```
17327 \cs_new:Npn \msg_expandable_error:nnnn #1#2#3#4
17328     { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} { } { } }
17329 \cs_new:Npn \msg_expandable_error:nnn #1#2#3
17330     { \msg_expandable_error:nnnnnn {#1} {#2} {#3} { } { } { } }
17331 \cs_new:Npn \msg_expandable_error:nn #1#2
17332     { \msg_expandable_error:nnnnnn {#1} {#2} { } { } { } { } }
17333 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnffff }
17334 \cs_generate_variant:Nn \msg_expandable_error:nnnnn  { nnfff }
17335 \cs_generate_variant:Nn \msg_expandable_error:nnnn   { nnff }
17336 \cs_generate_variant:Nn \msg_expandable_error:nnn    { nnf }
17337 \cs_new:Npn \__msg_expandable_error_module:nn #1#2
17338     {
17339        \exp_after:wN \exp_after:wN
17340        \exp_after:wN \use_none_delimit_by_q_stop:w
17341        \use:n { \::error ! ~ #2 : ~ #1 } \q_stop
17342     }
```

(*End definition for* `\msg_expandable_error:nnnnnn` *and others. These functions are documented on page 204.*)

34.15   Additions to **l3prg**

```
17343 ⟨@@=bool⟩
```

<code>\bool_lazy_all_p:n</code>
<code>\bool_lazy_all:nTF</code>
<code>__bool_lazy_all:n</code>

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.

```
17344 \prg_new_conditional:Npnn \bool_lazy_all:n #1 { p , T , F , TF }
17345     { \__bool_lazy_all:n #1 \q_recursion_tail \q_recursion_stop }
17346 \cs_new:Npn \__bool_lazy_all:n #1
17347     {
17348        \quark_if_recursion_tail_stop_do:nn {#1} { \prg_return_true: }
17349        \bool_if:nF {#1}
17350          { \use_i_delimit_by_q_recursion_stop:nw { \prg_return_false: } }
17351        \__bool_lazy_all:n
17352     }
```

(*End definition for* `\bool_lazy_all:nTF` *and* `\__bool_lazy_all:n`*. These functions are documented on page 205.*)

<code>\bool_lazy_and_p:nn</code>
<code>\bool_lazy_and:nnTF</code>

Only evaluate the second expression if the first is true.

```
17353 \prg_new_conditional:Npnn \bool_lazy_and:nn #1#2 { p , T , F , TF }
17354     {
17355        \bool_if:nTF {#1}
17356          { \bool_if:nTF {#2} { \prg_return_true: } { \prg_return_false: } }
17357          { \prg_return_false: }
17358     }
```

(*End definition for* `\bool_lazy_and:nnTF`*. This function is documented on page 205.*)

<code>\bool_lazy_any_p:n</code>
<code>\bool_lazy_any:nTF</code>
<code>__bool_lazy_any:n</code>

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.

```
17359 \prg_new_conditional:Npnn \bool_lazy_any:n #1 { p , T , F , TF }
17360     { \__bool_lazy_any:n #1 \q_recursion_tail \q_recursion_stop }
17361 \cs_new:Npn \__bool_lazy_any:n #1
17362     {
```

```
17363        \quark_if_recursion_tail_stop_do:nn {#1} { \prg_return_false: }
17364        \bool_if:nT {#1}
17365          { \use_i_delimit_by_q_recursion_stop:nw { \prg_return_true: } }
17366        \__bool_lazy_any:n
17367      }
```

(*End definition for* \bool_lazy_any:nTF *and* __bool_lazy_any:n. *These functions are documented on page 205.*)

\bool_lazy_or_p:nn   Only evaluate the second expression if the first is `false`.
\bool_lazy_or:nn*TF*
```
17368 \prg_new_conditional:Npnn \bool_lazy_or:nn #1#2 { p , T , F , TF }
17369      {
17370        \bool_if:nTF {#1}
17371          { \prg_return_true: }
17372          { \bool_if:nTF {#2} { \prg_return_true: } { \prg_return_false: } }
17373      }
```

(*End definition for* \bool_lazy_or:nnTF. *This function is documented on page 205.*)

\bool_log:N   Redirect output of \bool_show:N to the log.
\bool_log:c
\bool_log:n
```
17374 \cs_new_protected:Npn \bool_log:N
17375      { \__msg_log_next: \bool_show:N }
17376 \cs_new_protected:Npn \bool_log:n
17377      { \__msg_log_next: \bool_show:n }
17378 \cs_generate_variant:Nn \bool_log:N { c }
```

(*End definition for* \bool_log:N *and* \bool_log:n. *These functions are documented on page 205.*)

34.16 Additions to **l3prop**

```
17379 ⟨@@=prop⟩
```

\prop_count:N   Counting the key–value pairs in a property list is done using the same approach as for
\prop_count:c   other count functions: turn each entry into a `+1` then use integer evaluation to actually
__prop_count:nn   do the mathematics.
```
17380 \cs_new:Npn \prop_count:N #1
17381      {
17382        \int_eval:n
17383          {
17384            \c_zero
17385            \prop_map_function:NN #1 \__prop_count:nn
17386          }
17387      }
17388 \cs_new:Npn \__prop_count:nn #1#2 { + \c_one }
17389 \cs_generate_variant:Nn \prop_count:N { c }
```

(*End definition for* \prop_count:N *and* __prop_count:nn. *These functions are documented on page 206.*)

\prop_map_tokens:Nn   The mapping is very similar to \prop_map_function:NN. It grabs one key–value pair
\prop_map_tokens:cn   at a time, and stops when reaching the marker key \q_recursion_tail, which cannot
__prop_map_tokens:nwwn   appear in normal keys since those are strings. The odd construction \use:n {#1} allows
#1 to contain any token without interfering with \prop_map_break:. Argument #2 of
__prop_map_tokens:nwwn is \s__prop the first time, and is otherwise empty.

```
17390 \cs_new:Npn \prop_map_tokens:Nn #1#2
```

```
17391      {
17392        \exp_last_unbraced:Nno \__prop_map_tokens:nwwn {#2} #1
17393          \__prop_pair:wn \q_recursion_tail \s__prop { }
17394        \__prg_break_point:Nn \prop_map_break: { }
17395      }
17396    \cs_new:Npn \__prop_map_tokens:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
17397      {
17398        \if_meaning:w \q_recursion_tail #3
17399          \exp_after:wN \prop_map_break:
17400        \fi:
17401        \use:n {#1} {#3} {#4}
17402        \__prop_map_tokens:nwwn {#1}
17403      }
17404    \cs_generate_variant:Nn \prop_map_tokens:Nn { c }
```

(*End definition for* `\prop_map_tokens:Nn` *and* `\__prop_map_tokens:nwwn`. *These functions are documented on page 206.*)

\prop_log:N   Redirect output of \prop_show:N to the log.
\prop_log:c

```
17405    \cs_new_protected:Npn \prop_log:N
17406      { \__msg_log_next: \prop_show:N }
17407    \cs_generate_variant:Nn \prop_log:N { c }
```

(*End definition for* `\prop_log:N`. *This function is documented on page 206.*)

34.17 Additions to **l3seq**

```
17408    ⟨@@=seq⟩
```

\seq_mapthread_function:NNN   The idea is to first expand both sequences, adding the usual { ? __prg_break: } { }
\seq_mapthread_function:NcN   to the end of each one. This is most conveniently done in two steps using an auxiliary
\seq_mapthread_function:cNN   function. The mapping then throws away the first tokens of #2 and #5, which for items
\seq_mapthread_function:ccN   in the sequences will both be \s__seq __seq_item:n. The function to be mapped will
__seq_mapthread_function:wNN   then be applied to the two entries. When the code hits the end of one of the sequences,
__seq_mapthread_function:wNw   the break material will stop the entire loop and tidy up. This avoids needing to find the
__seq_mapthread_function:Nnnwnn   count of the two sequences, or worrying about which is longer.

```
17409    \cs_new:Npn \seq_mapthread_function:NNN #1#2#3
17410      { \exp_after:wN \__seq_mapthread_function:wNN #2 \q_stop #1 #3 }
17411    \cs_new:Npn \__seq_mapthread_function:wNN \s__seq #1 \q_stop #2#3
17412      {
17413        \exp_after:wN \__seq_mapthread_function:wNw #2 \q_stop #3
17414          #1 { ? \__prg_break: } { }
17415        \__prg_break_point:
17416      }
17417    \cs_new:Npn \__seq_mapthread_function:wNw \s__seq #1 \q_stop #2
17418      {
17419        \__seq_mapthread_function:Nnnwnn #2
17420          #1 { ? \__prg_break: } { }
17421        \q_stop
17422      }
17423    \cs_new:Npn \__seq_mapthread_function:Nnnwnn #1#2#3#4 \q_stop #5#6
17424      {
17425        \use_none:n #2
17426        \use_none:n #5
```

743

```
17427          #1 {#3} {#6}
17428          \__seq_mapthread_function:Nnnwnn #1 #4 \q_stop
17429        }
17430 \cs_generate_variant:Nn \seq_mapthread_function:NNN {     Nc }
17431 \cs_generate_variant:Nn \seq_mapthread_function:NNN { c , cc }
```

(*End definition for* `\seq_mapthread_function:NNN` *and others. These functions are documented on page* *206.*)

\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 x-expanding assignment.

```
17432 \cs_new_protected:Npn \seq_set_filter:NNn
17433   { \__seq_set_filter:NNNn \tl_set:Nx }
17434 \cs_new_protected:Npn \seq_gset_filter:NNn
17435   { \__seq_set_filter:NNNn \tl_gset:Nx }
17436 \cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
17437   {
17438     \__seq_push_item_def:n { \bool_if:nT {#4} { \__seq_wrap_item:n {##1} } }
17439     #1 #2 { #3 }
17440     \__seq_pop_item_def:
17441   }
```

(*End definition for* `\seq_set_filter:NNn`, `\seq_gset_filter:NNn`, *and* `\__seq_set_filter:NNNn`. *These functions are documented on page* *206.*)

\seq_set_map:NNn
\seq_gset_map:NNn
__seq_set_map:NNNn

Very similar to `\seq_set_filter:NNn`. We could actually merge the two within a single function, but it would have weird semantics.

```
17442 \cs_new_protected:Npn \seq_set_map:NNn
17443   { \__seq_set_map:NNNn \tl_set:Nx }
17444 \cs_new_protected:Npn \seq_gset_map:NNn
17445   { \__seq_set_map:NNNn \tl_gset:Nx }
17446 \cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
17447   {
17448     \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
17449     #1 #2 { #3 }
17450     \__seq_pop_item_def:
17451   }
```

(*End definition for* `\seq_set_map:NNn`, `\seq_gset_map:NNn`, *and* `\__seq_set_map:NNNn`. *These functions are documented on page* *207.*)

\seq_log:N
\seq_log:c

Redirect output of `\seq_show:N` to the log.

```
17452 \cs_new_protected:Npn \seq_log:N
17453   { \__msg_log_next: \seq_show:N }
17454 \cs_generate_variant:Nn \seq_log:N { c }
```

(*End definition for* `\seq_log:N`. *This function is documented on page* *207.*)

744

34.18 Additions to **l3skip**

\skip_split_finite_else_action:nnNN   This macro is useful when performing error checking in certain circumstances. If the ⟨*skip*⟩ register holds finite glue it sets #3 and #4 to the stretch and shrink component, resp. If it holds infinite glue set #3 and #4 to zero and issue the special action #2 which is probably an error message. Assignments are local.

```
17456 \cs_new:Npn \skip_split_finite_else_action:nnNN #1#2#3#4
17457   {
17458     \skip_if_finite:nTF {#1}
17459       {
17460         #3 = \etex_gluestretch:D #1 \scan_stop:
17461         #4 = \etex_glueshrink:D  #1 \scan_stop:
17462       }
17463       {
17464         #3 = \c_zero_skip
17465         #4 = \c_zero_skip
17466         #2
17467       }
17468   }
```

(*End definition for* \skip_split_finite_else_action:nnNN. *This function is documented on page 207.*)

\dim_log:N   Diagnostics. Redirect output of \dim_show:n to the log.
\dim_log:c
\dim_log:n
```
17469 \cs_new_eq:NN \dim_log:N \__kernel_register_log:N
17470 \cs_new_eq:NN \dim_log:c \__kernel_register_log:c
17471 \cs_new_protected:Npn \dim_log:n
17472   { \__msg_log_next: \dim_show:n }
```

(*End definition for* \dim_log:N *and* \dim_log:n. *These functions are documented on page 207.*)

\skip_log:N   Diagnostics. Redirect output of \skip_show:n to the log.
\skip_log:c
\skip_log:n
```
17473 \cs_new_eq:NN \skip_log:N \__kernel_register_log:N
17474 \cs_new_eq:NN \skip_log:c \__kernel_register_log:c
17475 \cs_new_protected:Npn \skip_log:n
17476   { \__msg_log_next: \skip_show:n }
```

(*End definition for* \skip_log:N *and* \skip_log:n. *These functions are documented on page 207.*)

\muskip_log:N   Diagnostics. Redirect output of \muskip_show:n to the log.
\muskip_log:c
\muskip_log:n
```
17477 \cs_new_eq:NN \muskip_log:N \__kernel_register_log:N
17478 \cs_new_eq:NN \muskip_log:c \__kernel_register_log:c
17479 \cs_new_protected:Npn \muskip_log:n
17480   { \__msg_log_next: \muskip_show:n }
```

(*End definition for* \muskip_log:N *and* \muskip_log:n. *These functions are documented on page 207.*)

34.19 Additions to l3tl

⟨@@=tl⟩

\tl_if_single_token_p:n
\tl_if_single_token:n*TF*

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.

```
17482 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
17483   {
17484     \tl_if_head_is_N_type:nTF {#1}
17485       { \__tl_if_empty_return:o { \use_none:n #1 } }
17486       {
17487         \tl_if_empty:nTF {#1}
17488           { \prg_return_false: }
17489           { \__tl_if_empty_return:o { \exp:w \exp_end_continue_f:w #1 } }
17490       }
17491   }
```

(*End definition for* \tl_if_single_token:nTF*. This function is documented on page 208.*)

\tl_reverse_tokens:n
__tl_reverse_group:nn

The same as \tl_reverse:n but with recursion within brace groups.

```
17492 \cs_new:Npn \tl_reverse_tokens:n #1
17493   {
17494     \etex_unexpanded:D \exp_after:wN
17495       {
17496         \exp:w
17497         \__tl_act:NNNnn
17498           \__tl_reverse_normal:nN
17499           \__tl_reverse_group:nn
17500           \__tl_reverse_space:n
17501           { }
17502           {#1}
17503       }
17504   }
17505 \cs_new:Npn \__tl_reverse_group:nn #1
17506   {
17507     \__tl_act_group_recurse:Nnn
17508       \__tl_act_reverse_output:n
17509       { \tl_reverse_tokens:n }
17510   }
```

__tl_act_group_recurse:Nnn

In many applications of __tl_act:NNNnn, we need to recursively apply some transformation within brace groups, then output. In this code, #1 is the output function, #2 is the transformation, which should expand in two steps, and #3 is the group.

```
17511 \cs_new:Npn \__tl_act_group_recurse:Nnn #1#2#3
17512   {
17513     \exp_args:Nf #1
17514       { \exp_after:wN \exp_after:wN \exp_after:wN { #2 {#3} } }
17515   }
```

(*End definition for* \tl_reverse_tokens:n *,* __tl_reverse_group:nn *, and* __tl_act_group_recurse:Nnn*. These functions are documented on page 208.*)

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`). Somewhat a hack!

```
17516 \cs_new:Npn \tl_count_tokens:n #1
17517   {
17518     \int_eval:n
17519       {
17520         \__tl_act:NNNnn
17521           \__tl_act_count_normal:nN
17522           \__tl_act_count_group:nn
17523           \__tl_act_count_space:n
17524           { }
17525           {#1}
17526       }
17527   }
17528 \cs_new:Npn \__tl_act_count_normal:nN #1 #2 { 1 + }
17529 \cs_new:Npn \__tl_act_count_space:n #1 { 1 + }
17530 \cs_new:Npn \__tl_act_count_group:nn #1 #2
17531   { 2 + \tl_count_tokens:n {#2} + }
```

(*End definition for* `\tl_count_tokens:n` *and others. These functions are documented on page 208.*)

The approach here is similar to that for doing a rescan, and so the same internals can be reused. Thus the plan is to insert a pair of tokens of the same charcode but different catcodes after the file has been read. This plus `\exp_not:N` allows the primitive to be used to carry out a set operation.

```
17532 \cs_new_protected:Npn \tl_set_from_file:Nnn
17533   { \__tl_set_from_file:NNnn \tl_set:Nn }
17534 \cs_new_protected:Npn \tl_gset_from_file:Nnn
17535   { \__tl_set_from_file:NNnn \tl_gset:Nn }
17536 \cs_generate_variant:Nn \tl_set_from_file:Nnn  { c }
17537 \cs_generate_variant:Nn \tl_gset_from_file:Nnn { c }
17538 \cs_new_protected:Npn \__tl_set_from_file:NNnn #1#2#3#4
17539   {
17540     \__file_if_exist:nT {#4}
17541       {
17542         \group_begin:
17543           \exp_args:No \etex_everyeof:D
17544             { \c__tl_rescan_marker_tl \exp_not:N }
17545           #3 \scan_stop:
17546           \exp_after:wN \__tl_from_file_do:w
17547           \exp_after:wN \prg_do_nothing:
17548             \tex_input:D \l__file_internal_name_tl \scan_stop:
17549         \exp_args:NNNo \group_end:
17550         #1 #2 \l__tl_internal_a_tl
17551       }
17552   }
17553 \exp_args:Nno \use:nn
17554   { \cs_set_protected:Npn \__tl_from_file_do:w #1 }
17555   { \c__tl_rescan_marker_tl }
17556   { \tl_set:No \l__tl_internal_a_tl {#1} }
```

(*End definition for* `\tl_set_from_file:Nnn` *and others. These functions are documented on page 211.*)

When reading a file and allowing expansion of the content, the set up only needs to prevent TeX complaining about the end of the file. That is done simply, with a group then used to trap the definition needed. Once the business is done using some scratch space, the tokens can be transferred to the real target.

```
17557 \cs_new_protected:Npn \tl_set_from_file_x:Nnn
17558   { \__tl_set_from_file_x:NNnn \tl_set:Nn }
17559 \cs_new_protected:Npn \tl_gset_from_file_x:Nnn
17560   { \__tl_set_from_file_x:NNnn \tl_gset:Nn }
17561 \cs_generate_variant:Nn \tl_set_from_file_x:Nnn  { c }
17562 \cs_generate_variant:Nn \tl_gset_from_file_x:Nnn { c }
17563 \cs_new_protected:Npn \__tl_set_from_file_x:NNnn #1#2#3#4
17564   {
17565     \__file_if_exist:nT {#4}
17566       {
17567         \group_begin:
17568           \etex_everyeof:D { \exp_not:N }
17569           #3 \scan_stop:
17570           \tl_set:Nx \l__tl_internal_a_tl
17571             { \tex_input:D \l__file_internal_name_tl \c_space_token }
17572         \exp_args:NNNo \group_end:
17573         #1 #2 \l__tl_internal_a_tl
17574       }
17575   }
```

(End definition for \tl_set_from_file_x:Nnn, \tl_gset_from_file_x:Nnn, *and* __tl_set_from_-
file_x:NNnn. *These functions are documented on page 211.)*

34.19.1 Unicode case changing

The mechanisms needed for case changing are somewhat involved, particularly to allow for all of the special cases. These functions also require the appropriate data extracted from the Unicode documentation (either manually or automatically).

Extra variants.

```
17576 \cs_generate_variant:Nn \tl_if_head_eq_catcode:nNTF { o }
```

(End definition for \tl_if_head_eq_catcode:oNTF. *This function is documented on page 99.)*

The user level functions here are all wrappers around the internal functions for case changing. Note that \tl_mixed_case:nn could be done without an internal, but this way the logic is slightly clearer as everything essentially follows the same path.

```
17577 \cs_new:Npn \tl_lower_case:n { \__tl_change_case:nnn { lower } { } }
17578 \cs_new:Npn \tl_upper_case:n { \__tl_change_case:nnn { upper } { } }
17579 \cs_new:Npn \tl_mixed_case:n { \__tl_mixed_case:nn { } }
17580 \cs_new:Npn \tl_lower_case:nn { \__tl_change_case:nnn { lower } }
17581 \cs_new:Npn \tl_upper_case:nn { \__tl_change_case:nnn { upper } }
17582 \cs_new:Npn \tl_mixed_case:nn { \__tl_mixed_case:nn }
```

(End definition for \tl_lower_case:n *and others. These functions are documented on page 208.)*

The mechanism for the core conversion of case is based on the idea that we can use a loop to grab the entire token list plus a quark: the latter is used as an end marker and to avoid any brace stripping. Depending on the nature of the first item in the grabbed argument, it can either processed as a single token, treated as a group or treated as a

748

space. These different cases all work by re-reading `#1` in the appropriate way, hence the repetition of `#1 \q_recursion_stop`.

```
17583 \cs_new:Npn \__tl_change_case:nnn #1#2#3
17584   {
17585     \etex_unexpanded:D \exp_after:wN
17586       {
17587         \exp:w
17588         \__tl_change_case_aux:nnn {#1} {#2} {#3}
17589       }
17590   }
17591 \cs_new:Npn \__tl_change_case_aux:nnn #1#2#3
17592   {
17593     \group_align_safe_begin:
17594     \__tl_change_case_loop:wnn
17595       #3 \q_recursion_tail \q_recursion_stop {#1} {#2}
17596     \__tl_change_case_result:n { }
17597   }
17598 \cs_new:Npn \__tl_change_case_loop:wnn #1 \q_recursion_stop
17599   {
17600     \tl_if_head_is_N_type:nTF {#1}
17601       { \__tl_change_case_N_type:Nwnn }
17602       {
17603         \tl_if_head_is_group:nTF {#1}
17604           { \__tl_change_case_group:nwnn }
17605           { \__tl_change_case_space:wnn }
17606       }
17607     #1 \q_recursion_stop
17608   }
```

Earlier versions of the code where only x-type expandable rather than f-type: this causes issues with nesting and so the slight performance hit is taken for a better outcome in usability terms. Setting up for f-type expandability has two requirements: a marker token after the main loop (see above) and a mechanism to "load" and finalise the result. That is handled in the code below, which includes the necessary material to end the `\exp:w` expansion.

```
17609 \cs_new:Npn \__tl_change_case_output:nwn #1#2 \__tl_change_case_result:n #3
17610   { #2 \__tl_change_case_result:n { #3 #1 } }
17611 \cs_generate_variant:Nn \__tl_change_case_output:nwn { V , o , v , f }
17612 \cs_new:Npn \__tl_change_case_end:wn #1 \__tl_change_case_result:n #2
17613   {
17614     \group_align_safe_end:
17615     \exp_end:
17616     #2
17617   }
```

Handling for the cases where the current argument is a brace group or a space is relatively easy. For the brace case, the routine works recursively, using the expandability of the mechanism to ensure that the result is finalised before storage. For the space case it is simply a question of removing the space in the input and storing it in the output. In both cases, and indeed for the N-type grabber, after removing the current item from the input `\__tl_change_case_loop:wnn` is inserted in front of the remaining tokens.

```
17618 \cs_new:Npn \__tl_change_case_group:nwnn #1#2 \q_recursion_stop #3#4
17619   {
17620     \__tl_change_case_output:own
```

```
17621          {
17622            \exp_after:wN
17623              {
17624                \exp:w
17625                \__tl_change_case_aux:nnn {#3} {#4} {#1}
17626              }
17627          }
17628        \__tl_change_case_loop:wnn #2 \q_recursion_stop {#3} {#4}
17629    }
17630  \exp_last_unbraced:NNo \cs_new:Npn \__tl_change_case_space:wnn \c_space_tl
17631    {
17632      \__tl_change_case_output:nwn { ~ }
17633      \__tl_change_case_loop:wnn
17634    }
```

For `N`-type arguments there are several stages to the approach. First, a simply check for the end-of-input marker, which if found triggers the final clean up and output step. Assuming that is not the case, the first check is for math-mode escaping: this test can encompass control sequences or other `N`-type tokens so is handled up front.

```
17635  \cs_new:Npn \__tl_change_case_N_type:Nwnn #1#2 \q_recursion_stop
17636    {
17637      \quark_if_recursion_tail_stop_do:Nn #1
17638        { \__tl_change_case_end:wn }
17639      \exp_after:wN \__tl_change_case_N_type:NNNnnn
17640        \exp_after:wN #1 \l_tl_case_change_math_tl
17641        \q_recursion_tail ? \q_recursion_stop {#2}
17642    }
```

Looking for math mode escape first requires a loop over the possible token pairs to see if the current input (`#1`) matches an open-math case (`#2`). If if does then this test loop is ended and a new input-gathering one is begun. The latter simply transfers material from the input to the output without any expansion, testing each `N`-type token to see if it matches the close-math case required. If that is the situation then the "math loop" stops and resumes the main loop: as that might be either the standard case-changing one or the mixed-case alternative, it is not hard-coded into the math loop but is rather passed as argument `#3` to `\__tl_change_case_math:NNNnnn`. If no close-math token is found then the final clean-up will be forced (*i.e.* there is no assumption of "well-behaved" code in terms of math mode).

```
17643  \cs_new:Npn \__tl_change_case_N_type:NNNnnn #1#2#3
17644    {
17645      \quark_if_recursion_tail_stop_do:Nn #2
17646        { \__tl_change_case_N_type:Nnnn #1 }
17647      \token_if_eq_meaning:NNTF #1 #2
17648        {
17649          \use_i_delimit_by_q_recursion_stop:nw
17650            {
17651              \__tl_change_case_math:NNNnnn
17652                #1 #3 \__tl_change_case_loop:wnn
17653            }
17654        }
17655        { \__tl_change_case_N_type:NNNnnn #1 }
17656    }
17657  \cs_new:Npn \__tl_change_case_math:NNNnnn #1#2#3#4
17658    {
```

```
17659        \__tl_change_case_output:nwn {#1}
17660        \__tl_change_case_math_loop:wNNnn #4 \q_recursion_stop #2 #3
17661      }
17662    \cs_new:Npn \__tl_change_case_math_loop:wNNnn #1 \q_recursion_stop
17663      {
17664        \tl_if_head_is_N_type:nTF {#1}
17665          { \__tl_change_case_math:NwNNnn }
17666          {
17667            \tl_if_head_is_group:nTF {#1}
17668              { \__tl_change_case_math_group:nwNNnn }
17669              { \__tl_change_case_math_space:wNNnn }
17670          }
17671        #1 \q_recursion_stop
17672      }
17673    \cs_new:Npn \__tl_change_case_math:NwNNnn #1#2 \q_recursion_stop #3#4
17674      {
17675        \token_if_eq_meaning:NNTF \q_recursion_tail #1
17676          { \__tl_change_case_end:wn }
17677          {
17678            \__tl_change_case_output:nwn {#1}
17679            \token_if_eq_meaning:NNTF #1 #3
17680              { #4 #2 \q_recursion_stop }
17681              { \__tl_change_case_math_loop:wNNnn #2 \q_recursion_stop #3#4 }
17682          }
17683      }
17684    \cs_new:Npn \__tl_change_case_math_group:nwNNnn #1#2 \q_recursion_stop
17685      {
17686        \__tl_change_case_output:nwn { {#1} }
17687        \__tl_change_case_math_loop:wNNnn #2 \q_recursion_stop
17688      }
17689    \exp_last_unbraced:NNo
17690      \cs_new:Npn \__tl_change_case_math_space:wNNnn \c_space_tl
17691      {
17692        \__tl_change_case_output:nwn { ~ }
17693        \__tl_change_case_math_loop:wNNnn
17694      }
```

Once potential math-mode cases are filtered out the next stage is to test if the token grabbed is a control sequence: they cannot be used in the lookup table and also may require expansion. At this stage the loop code starting `\__tl_change_case_loop:wnn` is inserted: all subsequent steps in the code which need a look-ahead are coded to rely on this and thus have `w`-type arguments if they may do a look-ahead.

```
17695    \cs_new:Npn \__tl_change_case_N_type:Nnnn #1#2#3#4
17696      {
17697        \token_if_cs:NTF #1
17698          { \__tl_change_case_cs_letterlike:Nnn #1 {#3} { } }
17699          { \__tl_change_case_char:Nnn #1 {#3} {#4} }
17700        \__tl_change_case_loop:wnn #2 \q_recursion_stop {#3} {#4}
17701      }
```

For character tokens there are some special cases to deal with then the majority of changes are covered by using the TEX data as a lookup along with expandable character generation. This avoids needing a very large number of macros or (as seen in earlier versions) a somewhat tricky split of the characters into various blocks. Notice that the

751

special case code may do a look-ahead so requires a final w-type argument whereas the core lookup table does not and also guarantees an output so f-type expansion may be used to obtain the case-changed result.

```
17702 \cs_new:Npn \__tl_change_case_char:Nnn #1#2#3
17703   {
17704     \cs_if_exist_use:cF { __tl_change_case_ #2 _ #3 :Nnw }
17705       { \use_ii:nn }
17706         #1
17707         {
17708           \use:c { __tl_change_case_ #2 _ sigma:Nnw } #1
17709             { \__tl_change_case_char:nN {#2} #1 }
17710         }
17711   }
```

For Unicode engines we can handle all characters directly. However, for the 8-bit engines the aim is to deal with (a subset of) Unicode (UTF-8) input. They deal with that by making the upper half of the range active, so we look for that and if found work out how many UTF-8 octets there are to deal with. Those can then be grabbed to reconstruct the full Unicode character, which is then used in a lookup. (As will become obvious below, there is no intention here of covering all of Unicode.)

```
17712 \cs_if_exist:NTF \utex_char:D
17713   {
17714     \cs_new:Npn \__tl_change_case_char:nN #1#2
17715       { \__tl_change_case_char_auxi:nN {#1} #2 }
17716   }
17717   {
17718     \cs_new:Npn \__tl_change_case_char:nN #1#2
17719       {
17720         \int_compare:nNnTF { `#2 } > { "80 }
17721           {
17722             \int_compare:nNnTF { `#2 } < { "E0 }
17723               { \__tl_change_case_char_UTFviii:nNNN {#1} #2 }
17724               {
17725                 \int_compare:nNnTF { `#2 } < { "F0 }
17726                   { \__tl_change_case_char_UTFviii:nNNNN {#1} #2 }
17727                   { \__tl_change_case_char_UTFviii:nNNNNN {#1} #2 }
17728               }
17729           }
17730           { \__tl_change_case_char_auxi:nN {#1} #2 }
17731       }
17732   }
17733 \cs_new:Npn \__tl_change_case_char_auxi:nN #1#2
17734   {
17735     \__tl_change_case_output:fwn
17736       {
17737         \cs_if_exist_use:cF { c__unicode_ #1 _ \token_to_str:N #2 _tl }
17738           { \__tl_change_case_char_auxii:nN {#1} #2 }
17739       }
17740   }
17741 \cs_if_exist:NTF \utex_char:D
17742   {
17743     \cs_new:Npn \__tl_change_case_char_auxii:nN #1#2
17744       {
17745         \int_compare:nNnTF { \use:c { __tl_lookup_ #1 :N } #2 } = { 0 }
```

```
17746              { \exp_stop_f: #2 }
17747              {
17748                \char_generate:nn
17749                  { \use:c { __tl_lookup_ #1 :N } #2 }
17750                  { \char_value_catcode:n { \use:c { __tl_lookup_ #1 :N } #2 } }
17751              }
17752          }
17753        \cs_new_protected:Npn \__tl_lookup_lower:N #1 { \tex_lccode:D `#1 }
17754        \cs_new_protected:Npn \__tl_lookup_upper:N #1 { \tex_uccode:D `#1 }
17755        \cs_new_eq:NN \__tl_lookup_title:N \__tl_lookup_upper:N
17756      }
17757      {
17758        \cs_new:Npn \__tl_change_case_char_auxii:nN #1#2 { \exp_stop_f: #2 }
17759        \cs_new:Npn \__tl_change_case_char_UTFviii:nNNN #1#2#3#4
17760          { \__tl_change_case_char_UTFviii:nnN {#1} {#2#4} #3 }
17761        \cs_new:Npn \__tl_change_case_char_UTFviii:nNNNN #1#2#3#4#5
17762          { \__tl_change_case_char_UTFviii:nnN {#1} {#2#4#5} #3 }
17763        \cs_new:Npn \__tl_change_case_char_UTFviii:nNNNNN #1#2#3#4#5#6
17764          { \__tl_change_case_char_UTFviii:nnN {#1} {#2#4#5#6} #3 }
17765        \cs_new:Npn \__tl_change_case_char_UTFviii:nnN #1#2#3
17766          {
17767            \cs_if_exist:cTF { c__unicode_ #1 _ \tl_to_str:n {#2} _tl }
17768              {
17769                \__tl_change_case_output:vwn
17770                  { c__unicode_ #1 _ \tl_to_str:n {#2} _tl }
17771              }
17772              { \__tl_change_case_output:nwn {#2} }
17773            #3
17774          }
17775      }
```

Before dealing with general control sequences there are the special ones to deal with. Letter-like control sequences are a simple look-up, while for accents the loop is much as done elsewhere. Notice that we have a no-op test to make sure there is no unexpected expansion of letter-like input. The third argument here is needed for mixed casing, where it if there is a hit there has to be a change-of-path.

```
17776  \cs_new:Npn \__tl_change_case_cs_letterlike:Nnn #1#2#3
17777    {
17778      \cs_if_exist:cTF { c__tl_change_case_ #2 _ \token_to_str:N #1 _tl }
17779        {
17780          \__tl_change_case_output:vwn
17781            { c__tl_change_case_ #2 _ \token_to_str:N #1 _tl }
17782          #3
17783        }
17784        {
17785          \cs_if_exist:cTF
17786            {
17787              c__tl_change_case_
17788              \str_if_eq:nnTF {#2} { lower } { upper } { lower }
17789              _ \token_to_str:N #1 _tl
17790            }
17791            {
17792              \__tl_change_case_output:nwn {#1}
17793              #3
```

```
17794                }
17795                {
17796                  \exp_after:wN \__tl_change_case_cs_accents:NN
17797                    \exp_after:wN #1 \l_tl_case_change_accents_tl
17798                    \q_recursion_tail \q_recursion_stop
17799                }
17800            }
17801        }
17802    \cs_new:Npn \__tl_change_case_cs_accents:NN #1#2
17803      {
17804        \quark_if_recursion_tail_stop_do:Nn #2
17805          { \__tl_change_case_cs:N #1 }
17806        \str_if_eq:nnTF {#1} {#2}
17807          {
17808            \use_i_delimit_by_q_recursion_stop:nw
17809              { \__tl_change_case_output:nwn {#1} }
17810          }
17811          { \__tl_change_case_cs_accents:NN #1 }
17812      }
```

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: that comes *after* the loop function which is therefore rearranged. In a LaTeX 2$_\varepsilon$ context, \protect needs to be treated specially, to prevent expansion of the next token but output it without braces.

```
17813    \cs_new:Npn \__tl_change_case_cs:N #1
17814      {
17815  ⟨*package⟩
17816        \str_if_eq:nnTF {#1} { \protect } { \__tl_change_case_protect:wNN }
17817  ⟨/package⟩
17818        \exp_after:wN \__tl_change_case_cs:NN
17819          \exp_after:wN #1 \l_tl_case_change_exclude_tl
17820          \q_recursion_tail \q_recursion_stop
17821      }
17822    \cs_new:Npn \__tl_change_case_cs:NN #1#2
17823      {
17824        \quark_if_recursion_tail_stop_do:Nn #2
17825          {
17826            \__tl_change_case_cs_expand:Nnw #1
17827              { \__tl_change_case_output:nwn {#1} }
17828          }
17829        \str_if_eq:nnTF {#1} {#2}
17830          {
17831            \use_i_delimit_by_q_recursion_stop:nw
17832              { \__tl_change_case_cs:NNn #1 }
17833          }
17834          { \__tl_change_case_cs:NN #1 }
17835      }
17836    \cs_new:Npn \__tl_change_case_cs:NNn #1#2#3
17837      {
17838        \__tl_change_case_output:nwn { #1 {#3} }
17839        #2
17840      }
17841  ⟨*package⟩
```

```
17842 \cs_new:Npn \__tl_change_case_protect:wNN #1 \q_recursion_stop #2 #3
17843   { \__tl_change_case_output:nwn { \protect #3 } #2 }
17844 ⟨/package⟩
```

When a control sequence is not on the exclude list the other test if to see if it is expandable. Once again, if there is a hit then the loop function is grabbed as part of the clean-up and reinserted before the now expanded material. The test for expandablity has to check for end-of-recursion as it is needed by the look-ahead code which might hit the end of the input. The test is done in two parts as `\bool_if:nTF` will choke if #1 is (!

```
17845 \cs_new:Npn \__tl_change_case_if_expandable:NTF #1
17846   {
17847     \token_if_expandable:NTF #1
17848       {
17849         \bool_if:nTF
17850           {
17851                 \token_if_protected_macro_p:N      #1
17852             || \token_if_protected_long_macro_p:N #1
17853             || \token_if_eq_meaning_p:NN \q_recursion_tail #1
17854           }
17855           { \use_ii:nn }
17856           { \use_i:nn }
17857       }
17858       { \use_ii:nn }
17859   }
17860 \cs_new:Npn \__tl_change_case_cs_expand:Nnw #1#2
17861   {
17862     \__tl_change_case_if_expandable:NTF #1
17863       { \__tl_change_case_cs_expand:NN #1 }
17864       { #2 }
17865   }
17866 \cs_new:Npn \__tl_change_case_cs_expand:NN #1#2
17867   { \exp_after:wN #2 #1 }
```

(*End definition for* `\__tl_change_case:nnn` *and others.*)

If the current char is an upper case 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.

```
17868 \cs_new:Npn \__tl_change_case_lower_sigma:Nnw #1#2#3#4 \q_recursion_stop
17869   {
17870     \int_compare:nNnTF { '#1 } = { "03A3 }
17871       {
17872         \__tl_change_case_output:fwn
17873           { \__tl_change_case_lower_sigma:w #4 \q_recursion_stop }
17874       }
17875       {#2}
17876     #3 #4 \q_recursion_stop
17877   }
17878 \cs_new:Npn \__tl_change_case_lower_sigma:w #1 \q_recursion_stop
17879   {
17880     \tl_if_head_is_N_type:nTF {#1}
17881       { \__tl_change_case_lower_sigma:Nw #1 \q_recursion_stop }
17882       { \c__unicode_final_sigma_tl }
17883   }
17884 \cs_new:Npn \__tl_change_case_lower_sigma:Nw #1#2 \q_recursion_stop
```

```
17885    {
17886      \__tl_change_case_if_expandable:NTF #1
17887        {
17888          \exp_after:wN \__tl_change_case_lower_sigma:w #1
17889            #2 \q_recursion_stop
17890        }
17891        {
17892          \token_if_letter:NTF #1
17893            { \c__unicode_std_sigma_tl }
17894            { \c__unicode_final_sigma_tl }
17895        }
17896    }
```

Simply skip to the final step for upper casing.

```
17897 \cs_new_eq:NN \__tl_change_case_upper_sigma:Nnw \use_ii:nn
```

(*End definition for* `\__tl_change_case_lower_sigma:Nnw` *and others.*)

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.

```
17898 \cs_if_exist:NTF \utex_char:D
17899    {
17900      \cs_new:Npn \__tl_change_case_lower_tr:Nnw #1#2
17901        {
17902          \int_compare:nNnTF { `#1 } = { "0049 }
17903            { \__tl_change_case_lower_tr_auxi:Nw }
17904            {
17905              \int_compare:nNnTF { `#1 } = { "0130 }
17906                { \__tl_change_case_output:nwn { i } }
17907                {#2}
17908            }
17909        }
```

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, which is done by the `\use_-i:nn` (it grabs `\__tl_change_case_loop:wn` and the dot-above char and discards the latter).

```
17910      \cs_new:Npn \__tl_change_case_lower_tr_auxi:Nw #1#2 \q_recursion_stop
17911        {
17912          \tl_if_head_is_N_type:nTF {#2}
17913            { \__tl_change_case_lower_tr_auxii:Nw #2 \q_recursion_stop }
17914            { \__tl_change_case_output:Vwn \c__unicode_dotless_i_tl }
17915          #1 #2 \q_recursion_stop
17916        }
17917      \cs_new:Npn \__tl_change_case_lower_tr_auxii:Nw #1#2 \q_recursion_stop
17918        {
17919          \__tl_change_case_if_expandable:NTF #1
17920            {
17921              \exp_after:wN \__tl_change_case_lower_tr_auxi:Nw #1
17922                #2 \q_recursion_stop
17923            }
17924            {
17925              \bool_if:nTF
```

```
17926                    {
17927                        \token_if_cs_p:N #1
17928                        || ! ( \int_compare_p:nNn { `#1 } = { "0307 } )
17929                    }
17930                    { \__tl_change_case_output:Vwn \c__unicode_dotless_i_tl }
17931                    {
17932                        \__tl_change_case_output:nwn { i }
17933                        \use_i:nn
17934                    }
17935            }
17936        }
17937    }
```

For 8-bit engines, dot-above is not available so there is a simple test for an upper-case I. Then we can look for the UTF-8 representation of an upper case dotted-I without the combining char. If it's not there, preserve the UTF-8 sequence as-is.

```
17938    {
17939        \cs_new:Npn \__tl_change_case_lower_tr:Nnw #1#2
17940            {
17941                \int_compare:nNnTF { `#1 } = { "0049 }
17942                    { \__tl_change_case_output:Vwn \c__unicode_dotless_i_tl }
17943                    {
17944                        \int_compare:nNnTF { `#1 } = { 196 }
17945                            { \__tl_change_case_lower_tr_auxi:Nw #1 {#2} }
17946                            {#2}
17947                    }
17948            }
17949        \cs_new:Npn \__tl_change_case_lower_tr_auxi:Nw #1#2#3#4
17950            {
17951                \int_compare:nNnTF { `#4 } = { 176 }
17952                    {
17953                        \__tl_change_case_output:nwn { i }
17954                        #3
17955                    }
17956                    {
17957                        #2
17958                        #3 #4
17959                    }
17960            }
17961    }
```

Upper casing is easier: just one exception with no context.

```
17962    \cs_new:Npn \__tl_change_case_upper_tr:Nnw #1#2
17963        {
17964            \int_compare:nNnTF { `#1 } = { "0069 }
17965                { \__tl_change_case_output:Vwn \c__unicode_dotted_I_tl }
17966                {#2}
17967        }
```

Straight copies.

```
17968    \cs_new_eq:NN \__tl_change_case_lower_az:Nnw \__tl_change_case_lower_tr:Nnw
17969    \cs_new_eq:NN \__tl_change_case_upper_az:Nnw \__tl_change_case_upper_tr:Nnw
```

(*End definition for* __tl_change_case_lower_tr:Nnw *and others.*)

For Lithuanian, the issue to be dealt with is dots over lower case letters: these should be present if there is another accent. That means that there is some work to do when lower casing I and J. The first step is a simple match attempt: `\c__tl_accents_lt_tl` contains accented upper case letters which should gain a dot-above char in their lower case form. This is done using `f`-type expansion so only one pass is needed to find if it works or not. If there was no hit, the second stage is to check for I, J and I-ogonek, and if the current char is a match to look for a following accent.

```
17970 \cs_new:Npn \__tl_change_case_lower_lt:Nnw #1
17971   {
17972     \exp_args:Nf \__tl_change_case_lower_lt:nNnw
17973       { \str_case:nVF #1 \c__unicode_accents_lt_tl \exp_stop_f: }
17974       #1
17975   }
17976 \cs_new:Npn \__tl_change_case_lower_lt:nNnw #1#2
17977   {
17978     \tl_if_blank:nTF {#1}
17979       {
17980         \exp_args:Nf \__tl_change_case_lower_lt:nnw
17981           {
17982             \int_case:nnF {`#2}
17983               {
17984                 { "0049 } i
17985                 { "004A } j
17986                 { "012E } \c__unicode_i_ogonek_tl
17987               }
17988             \exp_stop_f:
17989           }
17990       }
17991       {
17992         \__tl_change_case_output:nwn {#1}
17993         \use_none:n
17994       }
17995   }
17996 \cs_new:Npn \__tl_change_case_lower_lt:nnw #1#2
17997   {
17998     \tl_if_blank:nTF {#1}
17999       {#2}
18000       {
18001         \__tl_change_case_output:nwn {#1}
18002         \__tl_change_case_lower_lt:Nw
18003       }
18004   }
```

Grab the next char and see if it is one of the accents used in Lithuanian: if it is, add the dot-above char into the output.

```
18005 \cs_new:Npn \__tl_change_case_lower_lt:Nw #1#2 \q_recursion_stop
18006   {
18007     \tl_if_head_is_N_type:nT {#2}
18008       { \__tl_change_case_lower_lt:NNw }
18009     #1 #2 \q_recursion_stop
18010   }
18011 \cs_new:Npn \__tl_change_case_lower_lt:NNw #1#2#3 \q_recursion_stop
18012   {
18013     \__tl_change_case_if_expandable:NTF #2
```

758

```
18014        {
18015          \exp_after:wN \__tl_change_case_lower_lt:Nw \exp_after:wN #1 #2
18016            #3 \q_recursion_stop
18017        }
18018        {
18019          \bool_if:nT
18020            {
18021              ! \token_if_cs_p:N #2
18022              &&
18023              (
18024                    \int_compare_p:nNn { `#2 } = { "0300 }
18025                 || \int_compare_p:nNn { `#2 } = { "0301 }
18026                 || \int_compare_p:nNn { `#2 } = { "0303 }
18027              )
18028            }
18029            { \__tl_change_case_output:Vwn \c__unicode_dot_above_tl }
18030          #1 #2#3 \q_recursion_stop
18031        }
18032    }
```

For upper casing, the test required is for a dot-above char after an I, J or I-ogonek. First a test for the appropriate letter, and if found a look-ahead and potentially one token dropped.

```
18033 \cs_new:Npn \__tl_change_case_upper_lt:Nnw #1
18034   {
18035     \exp_args:Nf \__tl_change_case_upper_lt:nnw
18036       {
18037         \int_case:nnF {`#1}
18038           {
18039             { "0069 } I
18040             { "006A } J
18041             { "012F } \c__unicode_I_ogonek_tl
18042           }
18043         \exp_stop_f:
18044       }
18045   }
18046 \cs_new:Npn \__tl_change_case_upper_lt:nnw #1#2
18047   {
18048     \tl_if_blank:nTF {#1}
18049       {#2}
18050       {
18051         \__tl_change_case_output:nwn {#1}
18052         \__tl_change_case_upper_lt:Nw
18053       }
18054   }
18055 \cs_new:Npn \__tl_change_case_upper_lt:Nw #1#2 \q_recursion_stop
18056   {
18057     \tl_if_head_is_N_type:nT {#2}
18058       { \__tl_change_case_upper_lt:NNw }
18059     #1 #2 \q_recursion_stop
18060   }
18061 \cs_new:Npn \__tl_change_case_upper_lt:NNw #1#2#3 \q_recursion_stop
18062   {
18063     \__tl_change_case_if_expandable:NTF #2
```

```
18064        {
18065          \exp_after:wN \__tl_change_case_upper_lt:Nw \exp_after:wN #1 #2
18066            #3 \q_recursion_stop
18067        }
18068        {
18069          \bool_if:nTF
18070            {
18071                ! \token_if_cs_p:N #2
18072              && \int_compare_p:nNn { `#2 } = { "0307 }
18073            }
18074            { #1 }
18075            { #1 #2 }
18076          #3 \q_recursion_stop
18077        }
18078    }
```

(*End definition for* `\__tl_change_case_lower_lt:Nnw` *and others.*)

`\__tl_change_case_upper_de-alt:Nnw` A simple alternative version for German.

```
18079 \cs_new:cpn { __tl_change_case_upper_de-alt:Nnw } #1#2
18080    {
18081      \int_compare:nNnTF { `#1 } = { 223 }
18082        { \__tl_change_case_output:Vwn \c__unicode_upper_Eszett_tl }
18083        {#2}
18084    }
```

(*End definition for* `\__tl_change_case_upper_de-alt:Nnw.`)

`\__unicode_codepoint_to_UTFviii:n`
`\__unicode_codepoint_to_UTFviii_auxi:n`
`\__unicode_codepoint_to_UTFviii_auxii:Nnn`
`\__unicode_codepoint_to_UTFviii_auxiii:n`

This code will convert a codepoint into the correct UTF-8 representation. As there are a variable number of octets, the result starts with the numeral 1–4 to indicate the nature of the returned value. Note that this code will cover the full range even though at this stage it is not required here. Also note that longer-term this is likely to need a public interface and/or moving to l3str (see experimental string conversions). In terms of the algorithm itself, see https://en.wikipedia.org/wiki/UTF-8 for the octet pattern.

```
18085 \cs_new:Npn \__unicode_codepoint_to_UTFviii:n #1
18086    {
18087      \exp_args:Nf \__unicode_codepoint_to_UTFviii_auxi:n
18088        { \int_eval:n {#1} }
18089    }
18090 \cs_new:Npn \__unicode_codepoint_to_UTFviii_auxi:n #1
18091    {
18092      \if_int_compare:w #1 > "80 ~
18093        \if_int_compare:w #1 < "800 ~
18094          2
18095          \__unicode_codepoint_to_UTFviii_auxii:Nnn C {#1} { 64 }
18096          \__unicode_codepoint_to_UTFviii_auxiii:n {#1}
18097        \else:
18098          \if_int_compare:w #1 < "10000 ~
18099            3
18100            \__unicode_codepoint_to_UTFviii_auxii:Nnn E {#1} { 64 * 64 }
18101            \__unicode_codepoint_to_UTFviii_auxiii:n {#1}
18102            \__unicode_codepoint_to_UTFviii_auxiii:n
18103              { \int_div_truncate:nn {#1} { 64 } }
18104          \else:
```

```
18105              4
18106                \__unicode_codepoint_to_UTFviii_auxii:Nnn F
18107                  {#1} { 64 * 64 * 64 }
18108                \__unicode_codepoint_to_UTFviii_auxiii:n
18109                  { \int_div_truncate:nn {#1} { 64 * 64 } }
18110                \__unicode_codepoint_to_UTFviii_auxiii:n
18111                  { \int_div_truncate:nn {#1} { 64 } }
18112                \__unicode_codepoint_to_UTFviii_auxiii:n {#1}
18113
18114            \fi:
18115          \fi:
18116        \else:
18117          1 {#1}
18118        \fi:
18119    }
18120  \cs_new:Npn \__unicode_codepoint_to_UTFviii_auxii:Nnn #1#2#3
18121    { { \int_eval:n { "#10 + \int_div_truncate:nn {#2} {#3} } } }
18122  \cs_new:Npn \__unicode_codepoint_to_UTFviii_auxiii:n #1
18123    { { \int_eval:n { \int_mod:nn {#1} { 64 } + 128 } } }
```

(*End definition for* `\__unicode_codepoint_to_UTFviii:n` *and others.*)

`\c__unicode_std_sigma_tl` The above needs various special token lists containg pre-formed characters. This set are
`\c__unicode_final_sigma_tl` only available in Unicode engines, with no-op definitions for 8-bit use.
`\c__unicode_accents_lt_tl`
`\c__unicode_dot_above_tl`
`\c__unicode_upper_Eszett_tl`

```
18124  \cs_if_exist:NTF \utex_char:D
18125    {
18126      \tl_const:Nx \c__unicode_std_sigma_tl    { \utex_char:D "03C3 ~ }
18127      \tl_const:Nx \c__unicode_final_sigma_tl  { \utex_char:D "03C2 ~ }
18128      \tl_const:Nx \c__unicode_accents_lt_tl
18129        {
18130          \utex_char:D "00CC ~
18131            { \utex_char:D "0069 ~ \utex_char:D "0307 ~ \utex_char:D "0300 ~ }
18132          \utex_char:D "00CD ~
18133            { \utex_char:D "0069 ~ \utex_char:D "0307 ~ \utex_char:D "0301 ~ }
18134          \utex_char:D "0128 ~
18135            { \utex_char:D "0069 ~ \utex_char:D "0307 ~ \utex_char:D "0303 ~ }
18136        }
18137      \tl_const:Nx \c__unicode_dot_above_tl    { \utex_char:D "0307 ~ }
18138      \tl_const:Nx \c__unicode_upper_Eszett_tl { \utex_char:D "1E9E ~ }
18139    }
18140    {
18141      \tl_const:Nn \c__unicode_std_sigma_tl    { }
18142      \tl_const:Nn \c__unicode_final_sigma_tl  { }
18143      \tl_const:Nn \c__unicode_accents_lt_tl   { }
18144      \tl_const:Nn \c__unicode_dot_above_tl    { }
18145      \tl_const:Nn \c__unicode_upper_Eszett_tl { }
18146    }
```

(*End definition for* `\c__unicode_std_sigma_tl` *and others.*)

`\c__unicode_dotless_i_tl` For cases where there is an 8-bit option in the T1 font set up, a variant is provided in
`\c__unicode_dotted_I_tl` both cases.
`\c__unicode_i_ogonek_tl`
`\c__unicode_I_ogonek_tl`

```
18147  \group_begin:
18148    \cs_if_exist:NTF \utex_char:D
```

```
18149        {
18150          \cs_set_protected:Npn \__tl_tmp:w #1#2
18151            { \tl_const:Nx #1 { \utex_char:D "#2 ~ } }
18152        }
18153        {
18154          \cs_set_protected:Npn \__tl_tmp:w #1#2
18155            {
18156              \group_begin:
18157                \cs_set_protected:Npn \__tl_tmp:w ##1##2##3
18158                  {
18159                    \tl_const:Nx #1
18160                      {
18161                        \exp_after:wN \exp_after:wN \exp_after:wN
18162                          \exp_not:N \__char_generate:nn {##2} { 13 }
18163                        \exp_after:wN \exp_after:wN \exp_after:wN
18164                          \exp_not:N \__char_generate:nn {##3} { 13 }
18165                      }
18166                  }
18167                \tl_set:Nx \l__tl_internal_a_tl
18168                  { \__unicode_codepoint_to_UTFviii:n {"#2} }
18169                \exp_after:wN \__tl_tmp:w \l__tl_internal_a_tl
18170              \group_end:
18171            }
18172        }
18173      \__tl_tmp:w \c__unicode_dotless_i_tl { 0131 }
18174      \__tl_tmp:w \c__unicode_dotted_I_tl  { 0130 }
18175      \__tl_tmp:w \c__unicode_i_ogonek_tl  { 012F }
18176      \__tl_tmp:w \c__unicode_I_ogonek_tl  { 012E }
18177    \group_end:
```

(*End definition for* \c__unicode_dotless_i_tl *and others.*)

For 8-bit engines we now need to define the case-change data for the multi-octet mappings. These need a list of what code points are doable in T1 so the list is hard coded (there's no saving in loading the mappings dynamically). All of the straight-forward ones have two octets, so that is taken as read.

```
18178  \group_begin:
18179    \bool_if:nT
18180      {
18181        \sys_if_engine_pdftex_p: || \sys_if_engine_uptex_p:
18182      }
18183      {
18184        \cs_set_protected:Npn \__tl_loop:nn #1#2
18185          {
18186            \quark_if_recursion_tail_stop:n {#1}
18187            \tl_set:Nx \l__tl_internal_a_tl
18188              {
18189                \__unicode_codepoint_to_UTFviii:n {"#1}
18190                \__unicode_codepoint_to_UTFviii:n {"#2}
18191              }
18192            \exp_after:wN \__tl_tmp:w \l__tl_internal_a_tl
18193            \__tl_loop:nn
18194          }
18195        \cs_set_protected:Npn \__tl_tmp:w #1#2#3#4#5#6
18196          {
```

762

```
18197          \tl_const:cx
18198            {
18199              c__unicode_lower_
18200              \char_generate:nn {#2} { 12 }
18201              \char_generate:nn {#3} { 12 }
18202              _tl
18203            }
18204            {
18205              \exp_after:wN \exp_after:wN \exp_after:wN
18206                \exp_not:N \__char_generate:nn {#5} { 13 }
18207              \exp_after:wN \exp_after:wN \exp_after:wN
18208                \exp_not:N \__char_generate:nn {#6} { 13 }
18209            }
18210          \tl_const:cx
18211            {
18212              c__unicode_upper_
18213              \char_generate:nn {#5} { 12 }
18214              \char_generate:nn {#6} { 12 }
18215              _tl
18216            }
18217            {
18218              \exp_after:wN \exp_after:wN \exp_after:wN
18219                \exp_not:N \__char_generate:nn {#2} { 13 }
18220              \exp_after:wN \exp_after:wN \exp_after:wN
18221                \exp_not:N \__char_generate:nn {#3} { 13 }
18222            }
18223        }
18224      \__tl_loop:nn
18225        { 00C0 } { 00E0 }
18226        { 00C2 } { 00E2 }
18227        { 00C3 } { 00E3 }
18228        { 00C4 } { 00E4 }
18229        { 00C5 } { 00E5 }
18230        { 00C6 } { 00E6 }
18231        { 00C7 } { 00E7 }
18232        { 00C8 } { 00E8 }
18233        { 00C9 } { 00E9 }
18234        { 00CA } { 00EA }
18235        { 00CB } { 00EB }
18236        { 00CC } { 00EC }
18237        { 00CD } { 00ED }
18238        { 00CE } { 00EE }
18239        { 00CF } { 00EF }
18240        { 00D0 } { 00F0 }
18241        { 00D1 } { 00F1 }
18242        { 00D2 } { 00F2 }
18243        { 00D3 } { 00F3 }
18244        { 00D4 } { 00F4 }
18245        { 00D5 } { 00F5 }
18246        { 00D6 } { 00F6 }
18247        { 00D8 } { 00F8 }
18248        { 00D9 } { 00F9 }
18249        { 00DA } { 00FA }
18250        { 00DB } { 00FB }
```

```
18251        { 00DC } { 00FC }
18252        { 00DD } { 00FD }
18253        { 00DE } { 00FE }
18254        { 0100 } { 0101 }
18255        { 0102 } { 0103 }
18256        { 0104 } { 0105 }
18257        { 0106 } { 0107 }
18258        { 0108 } { 0109 }
18259        { 010A } { 010B }
18260        { 010C } { 010D }
18261        { 010E } { 010F }
18262        { 0110 } { 0111 }
18263        { 0112 } { 0113 }
18264        { 0114 } { 0115 }
18265        { 0116 } { 0117 }
18266        { 0118 } { 0119 }
18267        { 011A } { 011B }
18268        { 011C } { 011D }
18269        { 011E } { 011F }
18270        { 0120 } { 0121 }
18271        { 0122 } { 0123 }
18272        { 0124 } { 0125 }
18273        { 0128 } { 0129 }
18274        { 012A } { 012B }
18275        { 012C } { 012D }
18276        { 012E } { 012F }
18277        { 0132 } { 0133 }
18278        { 0134 } { 0135 }
18279        { 0136 } { 0137 }
18280        { 0139 } { 013A }
18281        { 013B } { 013C }
18282        { 013E } { 013F }
18283        { 0141 } { 0142 }
18284        { 0143 } { 0144 }
18285        { 0145 } { 0146 }
18286        { 0147 } { 0148 }
18287        { 014A } { 014B }
18288        { 014C } { 014D }
18289        { 014E } { 014F }
18290        { 0150 } { 0151 }
18291        { 0152 } { 0153 }
18292        { 0154 } { 0155 }
18293        { 0156 } { 0157 }
18294        { 0158 } { 0159 }
18295        { 015A } { 015B }
18296        { 015C } { 015D }
18297        { 015E } { 015F }
18298        { 0160 } { 0161 }
18299        { 0162 } { 0163 }
18300        { 0164 } { 0165 }
18301        { 0168 } { 0169 }
18302        { 016A } { 016B }
18303        { 016C } { 016D }
18304        { 016E } { 016F }
```

764

```
18305        { 0170 } { 0171 }
18306        { 0172 } { 0173 }
18307        { 0174 } { 0175 }
18308        { 0176 } { 0177 }
18309        { 0178 } { 00FF }
18310        { 0179 } { 017A }
18311        { 017B } { 017C }
18312        { 017D } { 017E }
18313        { 01CD } { 01CE }
18314        { 01CF } { 01D0 }
18315        { 01D1 } { 01D2 }
18316        { 01D3 } { 01D4 }
18317        { 01E2 } { 01E3 }
18318        { 01E6 } { 01E7 }
18319        { 01E8 } { 01E9 }
18320        { 01EA } { 01EB }
18321        { 01F4 } { 01F5 }
18322        { 0218 } { 0219 }
18323        { 021A } { 021B }
18324        \q_recursion_tail ?
18325        \q_recursion_stop
18326      \cs_set_protected:Npn \__tl_tmp:w #1#2#3
18327        {
18328          \group_begin:
18329            \cs_set_protected:Npn \__tl_tmp:w ##1##2##3
18330              {
18331                \tl_const:cx
18332                  {
18333                    c__unicode_ #3 _
18334                    \char_generate:nn {##2} { 12 }
18335                    \char_generate:nn {##3} { 12 }
18336                    _tl
18337                  }
18338                    {#2}
18339              }
18340            \tl_set:Nx \l__tl_internal_a_tl
18341              { \__unicode_codepoint_to_UTFviii:n { "#1 } }
18342            \exp_after:wN \__tl_tmp:w \l__tl_internal_a_tl
18343          \group_end:
18344        }
18345      \__tl_tmp:w { 00DF } { SS } { upper }
18346      \__tl_tmp:w { 00DF } { Ss } { title }
18347      \__tl_tmp:w { 0131 } { I }  { upper }
18348      }
18349    \group_end:
```

The (fixed) look-up mappings for letter-like control sequences.

```
18350  \group_begin:
18351    \cs_set_protected:Npn \__tl_change_case_setup:NN #1#2
18352      {
18353        \quark_if_recursion_tail_stop:N #1
18354        \tl_const:cn { c__tl_change_case_lower_ \token_to_str:N #1 _tl } { #2 }
18355        \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N #2 _tl } { #1 }
18356        \__tl_change_case_setup:NN
18357      }
```

```
18358     \__tl_change_case_setup:NN
18359     \AA \aa
18360     \AE \ae
18361     \DH \dh
18362     \DJ \dj
18363     \IJ \ij
18364     \L  \l
18365     \NG \ng
18366     \O  \o
18367     \OE \oe
18368     \SS \ss
18369     \TH \th
18370     \q_recursion_tail ?
18371     \q_recursion_stop
18372     \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N \i _tl } { I }
18373     \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N \j _tl } { J }
18374   \group_end:
```

\l_tl_case_change_accents_tl   A list of accents to leave alone.

```
18375 \tl_new:N \l_tl_case_change_accents_tl
18376 \tl_set:Nn \l_tl_case_change_accents_tl
18377   { \" \' \. \^ \` \~ \c \H \k \r \t \u \v }
```

(*End definition for* \l_tl_case_change_accents_tl. *This variable is documented on page 210.*)

__tl_mixed_case:nn
__tl_mixed_case_aux:nn
__tl_mixed_case_loop:wn
__tl_mixed_case_group:nwn
__tl_mixed_case_space:wn
__tl_mixed_case_N_type:Nwn
__tl_mixed_case_N_type:NNNnn
__tl_mixed_case_N_type:Nnn
__tl_mixed_case_letterlike:Nw
__tl_mixed_case_char:N
__tl_mixed_case_skip:N
__tl_mixed_case_skip:NN
__tl_mixed_case_skip_tidy:Nwn
__tl_mixed_case_char:nN

Mixed (title) casing requires some custom handling of the case changing of the first letter in the input followed by a switch to the normal lower casing routine. That could be covered by passing a set of functions to generic routines, but at the cost of making the process rather opaque. Instead, the approach taken here is to use a dedicated set of functions which keep the different loop requirements clearly separate.

The main loop looks for the first "real" char in the input (skipping any pre-letter chars). Once one is found, it is case changed to upper case but first checking that there is not an entry in the exceptions list. Note that simply grabbing the first token in the input is no good here: it can't handle pre-letter tokens or any special treatment of the first letter found (*e.g.* words starting with i in Turkish). Spaces at the start of the input are passed through without counting as being the "start" of the first word, while a brace group is assumed to be contain the first char with everything after the brace therefore lower cased.

```
18378 \cs_new:Npn \__tl_mixed_case:nn #1#2
18379   {
18380     \etex_unexpanded:D \exp_after:wN
18381       {
18382         \exp:w
18383         \__tl_mixed_case_aux:nn {#1} {#2}
18384       }
18385   }
18386 \cs_new:Npn \__tl_mixed_case_aux:nn #1#2
18387   {
18388     \group_align_safe_begin:
18389     \__tl_mixed_case_loop:wn
18390       #2 \q_recursion_tail \q_recursion_stop {#1}
18391     \__tl_change_case_result:n { }
18392   }
```

```
18393 \cs_new:Npn \__tl_mixed_case_loop:wn #1 \q_recursion_stop
18394   {
18395     \tl_if_head_is_N_type:nTF {#1}
18396       { \__tl_mixed_case_N_type:Nwn }
18397       {
18398         \tl_if_head_is_group:nTF {#1}
18399           { \__tl_mixed_case_group:nwn }
18400           { \__tl_mixed_case_space:wn }
18401       }
18402     #1 \q_recursion_stop
18403   }
18404 \cs_new:Npn \__tl_mixed_case_group:nwn #1#2 \q_recursion_stop #3
18405   {
18406     \__tl_change_case_output:own
18407       {
18408         \exp_after:wN
18409           {
18410             \exp:w
18411             \__tl_mixed_case_aux:nn {#3} {#1}
18412           }
18413       }
18414     \__tl_change_case_loop:wnn #2 \q_recursion_stop { lower } {#3}
18415   }
18416 \exp_last_unbraced:NNo \cs_new:Npn \__tl_mixed_case_space:wn \c_space_tl
18417   {
18418     \__tl_change_case_output:nwn { ~ }
18419     \__tl_mixed_case_loop:wn
18420   }
18421 \cs_new:Npn \__tl_mixed_case_N_type:Nwn #1#2 \q_recursion_stop
18422   {
18423     \quark_if_recursion_tail_stop_do:Nn #1
18424       { \__tl_change_case_end:wn }
18425     \exp_after:wN \__tl_mixed_case_N_type:NNNnn
18426       \exp_after:wN #1 \l_tl_case_change_math_tl
18427       \q_recursion_tail ? \q_recursion_stop {#2}
18428   }
18429 \cs_new:Npn \__tl_mixed_case_N_type:NNNnn #1#2#3
18430   {
18431     \quark_if_recursion_tail_stop_do:Nn #2
18432       { \__tl_mixed_case_N_type:Nnn #1 }
18433     \token_if_eq_meaning:NNTF #1 #2
18434       {
18435         \use_i_delimit_by_q_recursion_stop:nw
18436           {
18437             \__tl_change_case_math:NNNnnn
18438               #1 #3 \__tl_mixed_case_loop:wn
18439           }
18440       }
18441       { \__tl_mixed_case_N_type:NNNnn #1 }
18442   }
```

The business end of the loop is here: there is first a need to deal with any control sequence cases before looking for characters to skip. If there is a hit for a letter-like control sequence, switch to lower casing.

```
18443 \cs_new:Npn \__tl_mixed_case_N_type:Nnn #1#2#3
18444   {
18445     \token_if_cs:NTF #1
18446       {
18447         \__tl_change_case_cs_letterlike:Nnn #1 { upper }
18448           { \__tl_mixed_case_letterlike:Nw }
18449         \__tl_mixed_case_loop:wn #2 \q_recursion_stop {#3}
18450       }
18451       {
18452         \__tl_mixed_case_char:Nn #1 {#3}
18453         \__tl_change_case_loop:wnn #2 \q_recursion_stop { lower } {#3}
18454       }
18455   }
18456 \cs_new:Npn \__tl_mixed_case_letterlike:Nw #1#2 \q_recursion_stop
18457   { \__tl_change_case_loop:wnn #2 \q_recursion_stop { lower } }
```

As detailed above, handling a mixed case char means first looking for exceptions then treating as an upper cased letter, but with a list of tokens to skip over too.

```
18458 \cs_new:Npn \__tl_mixed_case_char:Nn #1#2
18459   {
18460     \cs_if_exist_use:cF { __tl_change_case_mixed_ #2 :Nnw }
18461       {
18462         \cs_if_exist_use:cF { __tl_change_case_upper_ #2 :Nnw }
18463           { \use_ii:nn }
18464       }
18465         #1
18466         { \__tl_mixed_case_skip:N #1 }
18467   }
18468 \cs_new:Npn \__tl_mixed_case_skip:N #1
18469   {
18470     \exp_after:wN \__tl_mixed_case_skip:NN
18471       \exp_after:wN #1 \l_tl_mixed_case_ignore_tl
18472       \q_recursion_tail \q_recursion_stop
18473   }
18474 \cs_new:Npn \__tl_mixed_case_skip:NN #1#2
18475   {
18476     \quark_if_recursion_tail_stop_do:nn {#2}
18477       { \__tl_mixed_case_char:N #1 }
18478     \int_compare:nNnT { `#1 }  = { `#2 }
18479       {
18480         \use_i_delimit_by_q_recursion_stop:nw
18481           {
18482             \__tl_change_case_output:nwn {#1}
18483             \__tl_mixed_case_skip_tidy:Nwn
18484           }
18485       }
18486     \__tl_mixed_case_skip:NN #1
18487   }
18488 \cs_new:Npn \__tl_mixed_case_skip_tidy:Nwn #1#2 \q_recursion_stop #3
18489   {
18490     \__tl_mixed_case_loop:wn #2 \q_recursion_stop
18491   }
18492 \cs_new:Npn \__tl_mixed_case_char:N #1
18493   {
```

```
18494        \cs_if_exist:cTF { c__unicode_title_  #1 _tl }
18495          {
18496            \__tl_change_case_output:fwn
18497              { \tl_use:c { c__unicode_title_ #1 _tl } }
18498          }
18499          { \__tl_change_case_char:nN { upper } #1 }
18500    }
```

(*End definition for* `\__tl_mixed_case:nn` *and others.*)

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.

```
18501  \cs_new:Npn \__tl_change_case_mixed_nl:Nnw #1
18502    {
18503      \bool_if:nTF
18504        {
18505              \int_compare_p:nNn { '#1 } = { 'i }
18506          || \int_compare_p:nNn { '#1 } = { 'I }
18507        }
18508        {
18509          \__tl_change_case_output:nwn { I }
18510          \__tl_change_case_mixed_nl:Nw
18511        }
18512    }
18513  \cs_new:Npn \__tl_change_case_mixed_nl:Nw #1#2 \q_recursion_stop
18514    {
18515      \tl_if_head_is_N_type:nT {#2}
18516        { \__tl_change_case_mixed_nl:NNw }
18517      #1 #2 \q_recursion_stop
18518    }
18519  \cs_new:Npn \__tl_change_case_mixed_nl:NNw #1#2#3 \q_recursion_stop
18520    {
18521      \__tl_change_case_if_expandable:NTF #2
18522        {
18523          \exp_after:wN \__tl_change_case_mixed_nl:Nw \exp_after:wN #1 #2
18524            #3 \q_recursion_stop
18525        }
18526        {
18527          \bool_if:nTF
18528            {
18529              ! ( \token_if_cs_p:N #2 )
18530              &&
18531                (
18532                      \int_compare_p:nNn { '#2 } = { 'j }
18533                  || \int_compare_p:nNn { '#2 } = { 'J }
18534                )
18535            }
18536            {
18537              \__tl_change_case_output:nwn { J }
18538              #1
18539            }
18540            { #1 #2 }
18541          #3 \q_recursion_stop
18542        }
18543    }
```

769

(*End definition for* __tl_change_case_mixed_nl:Nnw, __tl_change_case_mixed_nl:Nw, *and* __tl_-
change_case_mixed_nl:NNw.)

\l_tl_case_change_math_tl   The list of token pairs which are treated as math mode and so not case changed.

```
18544 \tl_new:N \l_tl_case_change_math_tl
18545 ⟨*package⟩
18546 \tl_set:Nn \l_tl_case_change_math_tl
18547   { $ $ \( \) }
18548 ⟨/package⟩
```

(*End definition for* \l_tl_case_change_math_tl. *This variable is documented on page 209.*)

\l_tl_case_change_exclude_tl   The list of commands for which an argument is not case changed.

```
18549 \tl_new:N \l_tl_case_change_exclude_tl
18550 ⟨*package⟩
18551 \tl_set:Nn \l_tl_case_change_exclude_tl
18552   { \cite \ensuremath \label \ref }
18553 ⟨/package⟩
```

(*End definition for* \l_tl_case_change_exclude_tl. *This variable is documented on page 209.*)

\l_tl_mixed_case_ignore_tl   Characters to skip over when finding the first letter in a word to be mixed cased.

```
18554 \tl_new:N \l_tl_mixed_case_ignore_tl
18555 \tl_set:Nx \l_tl_mixed_case_ignore_tl
18556   {
18557     ( % )
18558     [ % ]
18559     \cs_to_str:N \{ % \}
18560     ‘
18561     -
18562   }
```

(*End definition for* \l_tl_mixed_case_ignore_tl. *This variable is documented on page 210.*)

\tl_log:N   Redirect output of \tl_show:N to the log.
\tl_log:c
```
18563 \cs_new_protected:Npn \tl_log:N
18564   { \__msg_log_next: \tl_show:N }
18565 \cs_generate_variant:Nn \tl_log:N { c }
```

(*End definition for* \tl_log:N. *This function is documented on page 211.*)

\tl_log:n   Redirect output of \tl_show:n to the log.
```
18566 \cs_new_protected:Npn \tl_log:n
18567   { \__msg_log_next: \tl_show:n }
```

(*End definition for* \tl_log:n. *This function is documented on page 211.*)

34.20 Additions to **l3tokens**

<div style="float:left">

`\peek_N_type:`*TF*
`\_peek_execute_branches_N_type:`
`\__peek_N_type:w`
`\__peek_N_type_aux:nnw`

</div>

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, `\use_none_delimit_by_q_-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.

```
18569 \group_begin:
18570   \cs_set_protected:Npn \__peek_tmp:w #1 \q_stop
18571     {
18572       \cs_new_protected:Npn \__peek_execute_branches_N_type:
18573         {
18574           \if_int_odd:w
18575             \if_catcode:w \exp_not:N \l_peek_token {   \c_two \fi:
18576             \if_catcode:w \exp_not:N \l_peek_token }   \c_two \fi:
18577             \if_meaning:w \l_peek_token \c_space_token \c_two \fi:
18578             \c_one
18579           \exp_after:wN \__peek_N_type:w
18580             \token_to_meaning:N \l_peek_token
18581             \q_mark \__peek_N_type_aux:nnw
18582             #1 \q_mark \use_none_delimit_by_q_stop:w
18583             \q_stop
18584           \exp_after:wN \__peek_true:w
18585           \else:
18586             \exp_after:wN \__peek_false:w
18587           \fi:
18588         }
18589       \cs_new_protected:Npn \__peek_N_type:w ##1 #1 ##2 \q_mark ##3
18590         { ##3 {##1} {##2} }
18591     }
18592   \exp_after:wN \__peek_tmp:w \tl_to_str:n { outer } \q_stop
18593 \group_end:
18594 \cs_new_protected:Npn \__peek_N_type_aux:nnw #1 #2 #3 \fi:
18595   {
18596     \fi:
18597     \tl_if_in:noTF {#1} { \tl_to_str:n {ma} }
18598       { \__peek_true:w }
18599       { \tl_if_empty:nTF {#2} { \__peek_true:w } { \__peek_false:w } }
18600   }
18601 \cs_new_protected:Npn \peek_N_type:TF
18602   { \__peek_token_generic:NNTF \__peek_execute_branches_N_type: \scan_stop: }
18603 \cs_new_protected:Npn \peek_N_type:T
```

```
18604        { \__peek_token_generic:NNT \__peek_execute_branches_N_type: \scan_stop: }
18605   \cs_new_protected:Npn \peek_N_type:F
18606        { \__peek_token_generic:NNF \__peek_execute_branches_N_type: \scan_stop: }
```

(*End definition for* `\peek_N_type:TF` *and others. These functions are documented on page 212.*)

```
18607 ⟨/initex | package⟩
```

35   l3sys implementation

```
18608 ⟨*initex | package⟩
```

35.1   The name of the job

`\c_sys_jobname_str` Inherited from the LATEX3 name for the primitive: this needs to actually contain the text of the job name rather than the name of the primitive, of course.

```
18609 ⟨*initex⟩
18610 \tex_everyjob:D \exp_after:wN
18611   {
18612     \tex_the:D \tex_everyjob:D
18613     \str_const:Nx \c_sys_jobname_str { \tex_jobname:D }
18614   }
18615 ⟨/initex⟩
18616 ⟨*package⟩
18617 \str_const:Nx \c_sys_jobname_str { \tex_jobname:D }
18618 ⟨/package⟩
```

(*End definition for* `\c_sys_jobname_str`. *This variable is documented on page 213.*)

35.2   Time and date

`\c_sys_minute_int`   Copies of the information provided by TEX
`\c_sys_hour_int`
`\c_sys_day_int`
`\c_sys_month_int`
`\c_sys_year_int`

```
18619 \int_const:Nn \c_sys_minute_int
18620   { \int_mod:nn { \tex_time:D } { 60 } }
18621 \int_const:Nn \c_sys_hour_int
18622   { \int_div_truncate:nn { \tex_time:D } { 60 } }
18623 \int_const:Nn \c_sys_day_int    { \tex_day:D }
18624 \int_const:Nn \c_sys_month_int { \tex_month:D }
18625 \int_const:Nn \c_sys_year_int  { \tex_year:D }
```

(*End definition for* `\c_sys_minute_int` *and others. These variables are documented on page 213.*)

35.3   Detecting the engine

`\sys_if_engine_luatex_p:`   Set up the engine tests on the basis exactly one test should be true. Mainly a case of
`\sys_if_engine_luatex:TF`   looking for the appropriate marker primitive. For upTEX, there is a complexity in that
`\sys_if_engine_pdftex_p:`   setting `-kanji-internal=sjis` or `-kanji-internal=euc` effective makes it more like
`\sys_if_engine_pdftex:TF`   pTEX. In those cases we therefore report pTEX rather than upTEX.
`\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`

```
18626 \clist_map_inline:nn { lua , pdf , p , up , xe }
18627   {
18628     \cs_new_eq:cN { sys_if_engine_ #1 tex:T }  \use_none:n
18629     \cs_new_eq:cN { sys_if_engine_ #1 tex:F }  \use:n
18630     \cs_new_eq:cN { sys_if_engine_ #1 tex:TF } \use_ii:nn
18631     \cs_new_eq:cN { sys_if_engine_ #1 tex_p: } \c_false_bool
```

```
18632       }
18633 \cs_if_exist:NT \luatex_luatexversion:D
18634   {
18635     \cs_gset_eq:NN \sys_if_engine_luatex:T  \use:n
18636     \cs_gset_eq:NN \sys_if_engine_luatex:F  \use_none:n
18637     \cs_gset_eq:NN \sys_if_engine_luatex:TF \use_i:nn
18638     \cs_gset_eq:NN \sys_if_engine_luatex_p: \c_true_bool
18639     \str_const:Nn \c_sys_engine_str { luatex }
18640   }
18641 \cs_if_exist:NT \pdftex_pdftexversion:D
18642   {
18643     \cs_gset_eq:NN \sys_if_engine_pdftex:T  \use:n
18644     \cs_gset_eq:NN \sys_if_engine_pdftex:F  \use_none:n
18645     \cs_gset_eq:NN \sys_if_engine_pdftex:TF \use_i:nn
18646     \cs_gset_eq:NN \sys_if_engine_pdftex_p: \c_true_bool
18647     \str_const:Nn \c_sys_engine_str { pdftex }
18648   }
18649 \cs_if_exist:NT \ptex_kanjiskip:D
18650   {
18651     \bool_if:nTF
18652       {
18653         \cs_if_exist_p:N \uptex_disablecjktoken:D &&
18654         \int_compare_p:nNn { \ptex_jis:D "2121 } = { "3000 }
18655       }
18656       {
18657         \cs_gset_eq:NN \sys_if_engine_uptex:T  \use:n
18658         \cs_gset_eq:NN \sys_if_engine_uptex:F  \use_none:n
18659         \cs_gset_eq:NN \sys_if_engine_uptex:TF \use_i:nn
18660         \cs_gset_eq:NN \sys_if_engine_uptex_p: \c_true_bool
18661         \str_const:Nn \c_sys_engine_str { uptex }
18662       }
18663       {
18664         \cs_gset_eq:NN \sys_if_engine_ptex:T  \use:n
18665         \cs_gset_eq:NN \sys_if_engine_ptex:F  \use_none:n
18666         \cs_gset_eq:NN \sys_if_engine_ptex:TF \use_i:nn
18667         \cs_gset_eq:NN \sys_if_engine_ptex_p: \c_true_bool
18668         \str_const:Nn \c_sys_engine_str { ptex }
18669       }
18670   }
18671 \cs_if_exist:NT \xetex_XeTeXversion:D
18672   {
18673     \cs_gset_eq:NN \sys_if_engine_xetex:T  \use:n
18674     \cs_gset_eq:NN \sys_if_engine_xetex:F  \use_none:n
18675     \cs_gset_eq:NN \sys_if_engine_xetex:TF \use_i:nn
18676     \cs_gset_eq:NN \sys_if_engine_xetex_p: \c_true_bool
18677     \str_const:Nn \c_sys_engine_str { xetex }
18678   }
```

(*End definition for* `\sys_if_engine_luatex:TF` *and others. These functions are documented on page 213.*)

35.4   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.

```
18679 \bool_if:nTF
18680   {
18681        \cs_if_exist_p:N  \pdftex_pdfoutput:D
18682     && \int_compare_p:nNn \pdftex_pdfoutput:D > \c_zero
18683   }
18684   {
18685     \cs_new_eq:NN \sys_if_output_dvi:T  \use_none:n
18686     \cs_new_eq:NN \sys_if_output_dvi:F  \use:n
18687     \cs_new_eq:NN \sys_if_output_dvi:TF \use_ii:nn
18688     \cs_new_eq:NN \sys_if_output_dvi_p: \c_false_bool
18689     \cs_new_eq:NN \sys_if_output_pdf:T  \use:n
18690     \cs_new_eq:NN \sys_if_output_pdf:F  \use_none:n
18691     \cs_new_eq:NN \sys_if_output_pdf:TF \use_i:nn
18692     \cs_new_eq:NN \sys_if_output_pdf_p: \c_true_bool
18693     \str_const:Nn \c_sys_output_str { pdf }
18694   }
18695   {
18696     \cs_new_eq:NN \sys_if_output_dvi:T  \use:n
18697     \cs_new_eq:NN \sys_if_output_dvi:F  \use_none:n
18698     \cs_new_eq:NN \sys_if_output_dvi:TF \use_i:nn
18699     \cs_new_eq:NN \sys_if_output_dvi_p: \c_true_bool
18700     \cs_new_eq:NN \sys_if_output_pdf:T  \use_none:n
18701     \cs_new_eq:NN \sys_if_output_pdf:F  \use:n
18702     \cs_new_eq:NN \sys_if_output_pdf:TF \use_ii:nn
18703     \cs_new_eq:NN \sys_if_output_pdf_p: \c_false_bool
18704     \str_const:Nn \c_sys_output_str { dvi }
18705   }
```

(*End definition for* \sys_if_output_dvi:TF, \sys_if_output_pdf:TF, *and* \c_sys_output_str. *These functions are documented on page* 214.)

35.5 Deprecated functions

Deprecated 2015-09-07 for removal after 2016-12-31. The older logic supported only three engines so that has to be allowed for.

```
18706 \prg_new_eq_conditional:NNn \luatex_if_engine: \sys_if_engine_luatex:
18707   { T , F , TF , p }
18708 \prg_new_eq_conditional:NNn \xetex_if_engine:  \sys_if_engine_xetex:
18709   { T , F , TF , p }
18710 \bool_if:nTF
18711   {
18712     \sys_if_engine_luatex_p: ||
18713     \sys_if_engine_xetex_p:
18714   }
18715   {
18716     \cs_new_eq:NN \pdftex_if_engine:T  \use_none:n
18717     \cs_new_eq:NN \pdftex_if_engine:F  \use:n
18718     \cs_new_eq:NN \pdftex_if_engine:TF \use_ii:nn
18719     \cs_new_eq:NN \pdftex_if_engine_p: \c_false_bool
18720   }
18721   {
18722     \cs_new_eq:NN \pdftex_if_engine:T  \use:n
18723     \cs_new_eq:NN \pdftex_if_engine:F  \use_none:n
18724     \cs_new_eq:NN \pdftex_if_engine:TF \use_i:nn
```

`\cs_new_eq:NN \pdftex_if_engine_p: \c_true_bool`

`}`

Deprecated 2015-09-19 for removal after 2016-12-31.

`\cs_set_eq:NN \c_job_name_tl \c_sys_jobname_str`

⟨/initex | package⟩

36 l3luatex implementation

⟨*initex | package⟩

36.1 Breaking out to Lua

⟨*tex⟩

`\lua_now_x:n`
`\lua_now:n`
`\lua_shipout_x:n`
`\lua_shipout:n`
`\lua_escape_x:n`
`\lua_escape:n`

Wrappers around the primitives. 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/

`\cs_new:Npn \lua_now_x:n #1 { \luatex_directlua:D {#1} }`

`\cs_new:Npn \lua_now:n #1   { \lua_now_x:n { \exp_not:n {#1} } }`

`\cs_new_protected:Npn \lua_shipout_x:n #1 { \luatex_latelua:D {#1} }`

`\cs_new_protected:Npn \lua_shipout:n #1`

`  { \lua_shipout_x:n { \exp_not:n {#1} } }`

`\cs_new:Npn \lua_escape_x:n #1 { \luatex_luaescapestring:D {#1} }`

`\cs_new:Npn \lua_escape:n #1 { \lua_escape_x:n { \exp_not:n {#1} } }`

`\sys_if_engine_luatex:F`

`  {`

`    \clist_map_inline:nn`

`      { \lua_now_x:n , \lua_now:n , \lua_escape_x:n , \lua_escape:n }`

`      {`

`        \cs_set:Npn #1 ##1`

`          {`

`            \__msg_kernel_expandable_error:nnn`

`              { kernel } { luatex-required } { #1 }`

`          }`

`      }`

`    \clist_map_inline:nn`

`      { \lua_shipout_x :n , \lua_shipout:n }`

`      {`

`        \cs_set_protected:Npn #1 ##1`

`          {`

`            \__msg_kernel_error:nnn`

`              { kernel } { luatex-required } { #1 }`

`          }`

`      }`

`  }`

(*End definition for* `\lua_now_x:n` *and others. These functions are documented on page 215.*)

36.2 Messages

`\__msg_kernel_new:nnnn { kernel } { luatex-required }`

`  { LuaTeX~engine~not~in~use!~Ignoring~#1. }`

`  {`

```
18762    The~feature~you~are~using~is~only~available~
18763    with~the~LuaTeX~engine.~LaTeX3~ignored~'#1'.
18764  }
```

18765 ⟨/tex⟩

36.3 Lua functions for internal use

18766 ⟨*lua⟩

l3kernel Create a table for the kernel's own use.

```
18767 l3kernel = l3kernel or { }
```

(*End definition for* l3kernel.)

Various local copies of standard functions: naming convention is to retain the full text but replace all . by _.

```
18768 local tex_setcatcode    = tex.setcatcode
18769 local tex_sprint        = tex.sprint
18770 local tex_write         = tex.write
18771 local unicode_utf8_char = unicode.utf8.char
```

l3kernel.strcmp String comparison which gives the same results as pdfTeX's `\pdfstrcmp`, although the ordering should likely not be relied upon!

```
18772 local function strcmp (A, B)
18773   if A == B then
18774     tex_write("0")
18775   elseif A < B then
18776     tex_write("-1")
18777   else
18778     tex_write("1")
18779   end
18780 end
18781 l3kernel.strcmp = strcmp
```

(*End definition for* l3kernel.strcmp.)

l3kernel.charcat Creating arbitrary chars needs a category code table. As set up here, one may have been assigned earlier (see l3bootstrap) or a hard-coded one is used. The latter is intended for format mode and should be adjusted to match an eventual allocator.

```
18782 local charcat_table = l3kernel.charcat_table or 1
18783 local function charcat (charcode, catcode)
18784   tex_setcatcode(charcat_table, charcode, catcode)
18785   tex_sprint(charcat_table, unicode_utf8_char(charcode))
18786 end
18787 l3kernel.charcat = charcat
```

(*End definition for* l3kernel.charcat.)

18788 ⟨/lua⟩

18789 ⟨/initex | package⟩

36.4 Format mode code: font loader

⟨*fontloader⟩

In format mode, there needs to be a font loader available to let us use OpenType fonts. For testing, this is provided by `fontloader.lua` from the Speedata Publisher system (https://github.com/speedata/publisher). The code there is designed to be self-contained and has a certain number of build-in assumptions, so there is a small amount of compatibility required.

The code we load looks up `texmf` tree files using `kpse.filelist`, which isn't part of the standard `kpse` library. The interface is emulated using metatable.

```
18791 kpse.filelist = setmetatable({}, {
18792   __index = function (t, key)
18793     return kpse.lookup(key)
18794   end
18795 })
```

There is a built-in assumption in `fontloader.lua` that various environmental variables are set. We deal with that by intercepting the relevant names and returning something sane.

```
18796 local os_getenv = os.getenv
18797 function os.getenv (var)
18798   if var == "SP_FONT_PATH" then return "" end
18799   return os_getenv(var)
18800 end
```

As detailed in https://github.com/speedata/publisher/blob/develop/COPYING, the current license for Speedata Publisher is AGPLv3. We therefore only load the file and use its public interfaces rather than copying/modifying the code itself. Note though that we do have permission to use `fontloader.lua` as a public domain work (http://chat.stackexchange.com/transcript/message/27273687#27273687): if we want to develop a richer loader we may want to take advantage of that (which also applies to the simple shaper in the related `fonts.lua` file).

```
18801 local fontloader = require("fontloader.lua")
```

That done, register a callback which at present simply passes everything through. There's no attempt to pick up font settings (which presumably will be needed). Syntax is coerced to the same as for X⫟TEX.

```
18802 callback.register("define_font",
18803   function (name, size, id)
18804     local opts, opttab, otfeatures = "", { }, { }
18805     if string.match(name, "^%[") then
18806       name, opts = string.match(name, "^%[([^%]]*)%][^:]*:?(.*)")
18807     end
18808     if opts ~= "" then
18809       for _,kv in ipairs(string.explode(opts,";")) do
18810         if string.match(kv, "=") then
18811           local k, v = string.match(kv, "([^=]*)=?(.*)")
18812           opttab[k] = v
18813         else
18814           if string.match(kv, "^+") then
18815             otfeatures[string.sub(kv,2,-1)] = "true"
18816           elseif string.match(kv, "^-") then
18817             otfeatures[string.sub(kv,2,-1)] = "false"
18818           else
```

```
18819            otfeatures[kv] = "true"
18820          end
18821        end
18822      end
18823    end
18824    if next(otfeatures) then
18825      opttab["otfeatures"] = otfeatures
18826    end
18827    return select(2, fontloader.define_font(name, size, opttab))
18828  end
18829 )
```

18830 ⟨/fontloader⟩

37 l3drivers Implementation

18831 ⟨*initex | package⟩

18832 ⟨@@=driver⟩

Whilst there is a reasonable amount of code overlap between drivers, it is much clearer to have the blocks more-or-less separated than run in together and DocStripped out in parts. As such, most of the following is set up on a per-driver basis, though there is some common code (again given in blocks not interspersed with other material).

All the file identifiers are up-front so that they come out in the right place in the files.

18833 ⟨*package⟩
18834 \ProvidesExplFile
18835 ⟨*dvipdfmx⟩
18836   {l3dvidpfmx.def}{\ExplFileDate}{\ExplFileVersion}
18837   {L3 Experimental driver: dvipdfmx}
18838 ⟨/dvipdfmx⟩
18839 ⟨*dvips⟩
18840   {l3dvips.def}{\ExplFileDate}{\ExplFileVersion}
18841   {L3 Experimental driver: dvips}
18842 ⟨/dvips⟩
18843 ⟨*dvisvgm⟩
18844   {l3dvisvgm.def}{\ExplFileDate}{\ExplFileVersion}
18845   {L3 Experimental driver: dvisvgm}
18846 ⟨/dvisvgm⟩
18847 ⟨*pdfmode⟩
18848   {l3pdfmode.def}{\ExplFileDate}{\ExplFileVersion}
18849   {L3 Experimental driver: PDF mode}
18850 ⟨/pdfmode⟩
18851 ⟨*xdvipdfmx⟩
18852   {l3xdvidpfmx.def}{\ExplFileDate}{\ExplFileVersion}
18853   {L3 Experimental driver: xdvipdfmx}
18854 ⟨/xdvipdfmx⟩
18855 ⟨/package⟩

37.1 pdfmode driver

18856 ⟨*pdfmode⟩

The direct PDF driver covers both pdfTeX and LuaTeX. The latter renames/restructures the driver primitives but this can be handled at one level of abstraction. As such, we

avoid using two separate drivers for this material at the cost of some x-type definitions to get everything expanded up-front.

37.1.1 Basics

__driver_literal:n This is equivalent to `\special{pdf:}` but the engine can track it. Without the `direct` keyword everything is kept in sync: the transformation matrix is set to the current point automatically. Note that this is still inside the text (BT . . . ET block).

```
18857 \cs_new_protected:Npx \__driver_literal:n #1
18858   {
18859     \cs_if_exist:NTF \luatex_pdfextension:D
18860       { \luatex_pdfextension:D literal }
18861       { \pdftex_pdfliteral:D }
18862         {#1}
18863   }
```

(*End definition for* __driver_literal:n.)

__driver_scope_begin: Higher-level interfaces for saving and restoring the graphic state.
__driver_scope_end:

```
18864 \cs_new_protected:Npx \__driver_scope_begin:
18865   {
18866     \cs_if_exist:NTF \luatex_pdfextension:D
18867       { \luatex_pdfextension:D save \scan_stop: }
18868       { \pdftex_pdfsave:D }
18869   }
18870 \cs_new_protected:Npx \__driver_scope_end:
18871   {
18872     \cs_if_exist:NTF \luatex_pdfextension:D
18873       { \luatex_pdfextension:D restore \scan_stop: }
18874       { \pdftex_pdfrestore:D }
18875   }
```

(*End definition for* __driver_scope_begin: *and* __driver_scope_end:.)

__driver_matrix:n Here the appropriate function is set up to insert an affine matrix into the PDF. With pdfTEX and LuaTEX in direct PDF output mode there is a primitive for this, which only needs the rotation/scaling/skew part.

```
18876 \cs_new_protected:Npx \__driver_matrix:n #1
18877   {
18878     \cs_if_exist:NTF \luatex_pdfextension:D
18879       { \luatex_pdfextension:D setmatrix }
18880       { \pdftex_pdfsetmatrix:D }
18881         {#1}
18882   }
```

(*End definition for* __driver_matrix:n.)

37.1.2 Color

\l__driver_current_color_tl The current color in driver-dependent format: pick up the package-mode data if available.

```
18883 \tl_new:N \l__driver_current_color_tl
18884 \tl_set:Nn \l__driver_current_color_tl { 0~g~0~G }
18885 ⟨*package⟩
18886 \AtBeginDocument
18887   {
```

779

```
18888          \@ifpackageloaded { color }
18889            { \tl_set:Nn \l__driver_current_color_tl { \current@color } }
18890            { }
18891      }
18892 ⟨/package⟩
```

(*End definition for* \l__driver_current_color_tl.)

\l__driver_color_stack_int pdfTeX and LuaTeX have multiple stacks available, and to track which one is in use a variable is required.

```
18893 \int_new:N \l__driver_color_stack_int
```

(*End definition for* \l__driver_color_stack_int.)

_driver_color_ensure_current: There is a dedicated primitive/primitive interface for setting colors. As with scoping, __driver_color_reset: this approach is not suitable for cached operations.

```
18894 \cs_new_protected:Npx \__driver_color_ensure_current:
18895   {
18896      \cs_if_exist:NTF \luatex_pdfextension:D
18897        { \luatex_pdfextension:D colorstack }
18898        { \pdftex_pdfcolorstack:D }
18899          \exp_not:N \l__driver_color_stack_int push
18900            { \exp_not:N \l__driver_current_color_tl }
18901      \group_insert_after:N \exp_not:N \__driver_color_reset:
18902   }
18903 \cs_new_protected:Npx \__driver_color_reset:
18904   {
18905      \cs_if_exist:NTF \luatex_pdfextension:D
18906        { \luatex_pdfextension:D colorstack }
18907        { \pdftex_pdfcolorstack:D }
18908          \exp_not:N \l__driver_color_stack_int pop \scan_stop:
18909   }
```

(*End definition for* __driver_color_ensure_current: *and* __driver_color_reset:.)

```
18910 ⟨/pdfmode⟩
```

37.2 dvipdfmx driver

```
18911 ⟨*dvipdfmx | xdvipdfmx⟩
```

The dvipdfmx shares code with the PDF mode one (using the common section to this file) but also with xdvipdfmx. The latter is close to identical to dvipdfmx and so all of the code here is extracted for both drivers, with some clean up for xdvipdfmx as required.

37.2.1 Basics

__driver_literal:n Equivalent to pdf:content but favoured as the link to the pdfTeX primitive approach is clearer.

```
18912 \cs_new_protected:Npn \__driver_literal:n #1
18913   { \tex_special:D { pdf:literal~ #1 } }
```

(*End definition for* __driver_literal:n.)

$\__driver_scope_begin:$
$\__driver_scope_end:$

Scoping is done using direct PDF operations here.

```
18914 \cs_new_protected:Npn \__driver_scope_begin:
18915   { \__driver_literal:n { q } }
18916 \cs_new_protected:Npn \__driver_scope_end:
18917   { \__driver_literal:n { Q } }
```

(*End definition for* $\__driver_scope_begin:$ *and* $\__driver_scope_end:$.)

$\__driver_matrix:n$

With (x)dvipdfmx the matrix has to include a translation part: that is always zero and so is built in here so that the same internal interface works for all PDF-related drivers.

```
18918 \cs_new_protected:Npn \__driver_matrix:n #1
18919   { \__driver_literal:n { #1 \c_space_tl 0~0~cm } }
```

(*End definition for* $\__driver_matrix:n$.)

37.2.2 Color

$\l__driver_current_color_tl$

The current color in driver-dependent format.

```
18920 \tl_new:N \l__driver_current_color_tl
18921 \tl_set:Nn \l__driver_current_color_tl { [ 0 ] }
18922 ⟨*package⟩
18923 \AtBeginDocument
18924   {
18925     \@ifpackageloaded { color }
18926       { \tl_set:Nn \l__driver_current_color_tl { \current@color } }
18927       { }
18928   }
18929 ⟨/package⟩
```

(*End definition for* $\l__driver_current_color_tl$.)

$\__driver_color_ensure_current:$
$\__driver_color_reset:$

Directly set the color using the specials with optimisation support.

```
18930 \cs_new_protected:Npn \__driver_color_ensure_current:
18931   {
18932     \tex_special:D { pdf:bcolor~\l__driver_current_color_tl }
18933     \group_insert_after:N \__driver_color_reset:
18934   }
18935 \cs_new_protected:Npn \__driver_color_reset:
18936   { \tex_special:D { pdf:ecolor } }
```

(*End definition for* $\__driver_color_ensure_current:$ *and* $\__driver_color_reset:$.)

```
18937 ⟨/dvipdfmx | xdvipdfmx⟩
```

37.3 xdvipdfmx driver

```
18938 ⟨*xdvipdfmx⟩
```

37.3.1 Color

$\__driver_color_ensure_current:$
$\__driver_color_reset:$

The LaTeX2_ε driver uses dvips-like specials so there has to be a change of set up if color is loaded.

```
18939 ⟨*package⟩
18940 \AtBeginDocument
18941   {
```

```
18942        \@ifpackageloaded { color }
18943          {
18944            \cs_set_protected:Npn \__driver_color_ensure_current:
18945              {
18946                \tex_special:D { color~push~\l__driver_current_color_tl }
18947                \group_insert_after:N \__driver_color_reset:
18948              }
18949            \cs_set_protected:Npn \__driver_color_reset:
18950              { \tex_special:D { color~pop } }
18951          }
18952          { }
18953      }
18954 ⟨/package⟩
```

(*End definition for* `\__driver_color_ensure_current:` *and* `\__driver_color_reset:`.)

```
18955 ⟨/xdvipdfmx⟩
```

37.4 Common code for PDF production

As all of the drivers which understand PDF-targeted specials act in much the same way there is a lot of shared code. Rather than try to DocStrip it interspersed with the above, we collect all of it here.

```
18956 ⟨*dvipdfmx | pdfmode | xdvipdfmx⟩
```

37.4.1 Box operations

`\__driver_box_use_clip:N` The general method is to save the current location, define a clipping path equivalent to the bounding box, then insert the content at the current position and in a zero width box. The "real" width is then made up using a horizontal skip before tidying up. There are other approaches that can be taken (for example using XForm objects), but the logic here shares as much code as possible and uses the same conversions (and so same rounding errors) in all cases.

```
18957 \cs_new_protected:Npn \__driver_box_use_clip:N #1
18958   {
18959     \__driver_scope_begin:
18960     \__driver_literal:n
18961       {
18962         0~
18963         \dim_to_decimal_in_bp:n { -\box_dp:N #1 } ~
18964         \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
18965         \dim_to_decimal_in_bp:n { \box_ht:N #1 + \box_dp:N #1 } ~
18966         re~W~n
18967       }
18968     \hbox_overlap_right:n { \box_use:N #1 }
18969     \__driver_scope_end:
18970     \skip_horizontal:n { \box_wd:N #1 }
18971   }
```

(*End definition for* `\__driver_box_use_clip:N`.)

`\__driver_box_use_rotate:Nn`
`\l__driver_cos_fp`
`\l__driver_sin_fp`
Rotations are set using an affine transformation matrix which therefore requires sine/cosine values not the angle itself. We store the rounded values to avoid rounding twice. There are also a couple of comparisons to ensure that `-0` is not written to the

output, as this avoids any issues with problematic display programs. Note that numbers are compared to 0 after rounding.

```
18972 \cs_new_protected:Npn \__driver_box_use_rotate:Nn #1#2
18973   {
18974     \__driver_scope_begin:
18975     \box_set_wd:Nn #1 \c_zero_dim
18976     \fp_set:Nn \l__driver_cos_fp { round ( cosd ( #2 ) , 5 ) }
18977     \fp_compare:nNnT \l__driver_cos_fp = \c_zero_fp
18978       { \fp_zero:N \l__driver_cos_fp }
18979     \fp_set:Nn \l__driver_sin_fp { round ( sind ( #2 ) , 5 ) }
18980     \__driver_matrix:n
18981       {
18982         \fp_use:N \l__driver_cos_fp \c_space_tl
18983         \fp_compare:nNnTF \l__driver_sin_fp = \c_zero_fp
18984           { 0~0 }
18985           {
18986             \fp_use:N \l__driver_sin_fp
18987             \c_space_tl
18988             \fp_eval:n { -\l__driver_sin_fp }
18989           }
18990         \c_space_tl
18991         \fp_use:N \l__driver_cos_fp
18992       }
18993     \box_use:N #1
18994     \__driver_scope_end:
18995   }
18996 \fp_new:N \l__driver_cos_fp
18997 \fp_new:N \l__driver_sin_fp
```

(*End definition for* __driver_box_use_rotate:Nn, \l__driver_cos_fp, *and* \l__driver_sin_fp.)

__driver_box_use_scale:Nnn The same idea as for rotation but without the complexity of signs and cosines.

```
18998 \cs_new_protected:Npn \__driver_box_use_scale:Nnn #1#2#3
18999   {
19000     \__driver_scope_begin:
19001     \__driver_matrix:n
19002       {
19003         \fp_eval:n { round ( #2 , 5 ) } ~
19004         0~0~
19005         \fp_eval:n { round ( #3 , 5 ) }
19006       }
19007     \hbox_overlap_right:n { \box_use:N #1 }
19008     \__driver_scope_end:
19009   }
```

(*End definition for* __driver_box_use_scale:Nnn.)

37.5 Drawing

__driver_draw_literal:n Pass data through using a dedicated interface.
__driver_draw_literal:x

```
19010 \cs_new_eq:NN \__driver_draw_literal:n \__driver_literal:n
19011 \cs_generate_variant:Nn \__driver_draw_literal:n { x }
```

(*End definition for* __driver_draw_literal:n.)

`\__driver_draw_begin:`
`\__driver_draw_end:`

No special requirements here, so simply set up a drawing scope.

```
19012 \cs_new_protected:Npn \__driver_draw_begin:
19013   { \__driver_draw_scope_begin: }
19014 \cs_new_protected:Npn \__driver_draw_end:
19015   { \__driver_draw_scope_end: }
```

(*End definition for* `\__driver_draw_begin:` *and* `\__driver_draw_end:`.)

`\__driver_draw_scope_begin:`
`\__driver_draw_scope_end:`

In contrast to a general scope, a drawing scope is always done using the PDF operators so is the same for all relevant drivers.

```
19016 \cs_new_protected:Npn \__driver_draw_scope_begin:
19017   { \__driver_draw_literal:n { q } }
19018 \cs_new_protected:Npn \__driver_draw_scope_end:
19019   { \__driver_draw_literal:n { Q } }
```

(*End definition for* `\__driver_draw_scope_begin:` *and* `\__driver_draw_scope_end:`.)

`\__driver_draw_moveto:nn`
`\__driver_draw_lineto:nn`
`\__driver_draw_curveto:nnnnnn`
`\__driver_draw_rectangle:nnnn`

Path creation operations all resolve directly to PDF primitive steps, with only the need to convert to `bp`. Notice that x-type expansion is included here to ensure that any variable values are forced to literals before any possible caching.

```
19020 \cs_new_protected:Npn \__driver_draw_moveto:nn #1#2
19021   {
19022     \__driver_draw_literal:x
19023       { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ m }
19024   }
19025 \cs_new_protected:Npn \__driver_draw_lineto:nn #1#2
19026   {
19027     \__driver_draw_literal:x
19028       { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ l }
19029   }
19030 \cs_new_protected:Npn \__driver_draw_curveto:nnnnnn #1#2#3#4#5#6
19031   {
19032     \__driver_draw_literal:x
19033       {
19034         \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
19035         \dim_to_decimal_in_bp:n {#3} ~ \dim_to_decimal_in_bp:n {#4} ~
19036         \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
19037         c
19038       }
19039   }
19040 \cs_new_protected:Npn \__driver_draw_rectangle:nnnn #1#2#3#4
19041   {
19042     \__driver_draw_literal:x
19043       {
19044         \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
19045         \dim_to_decimal_in_bp:n {#3} ~ \dim_to_decimal_in_bp:n {#4} ~
19046         re
19047       }
19048   }
```

(*End definition for* `\__driver_draw_moveto:nn` *and others.*)

`\__driver_draw_evenodd_rule:`
`\__driver_draw_nonzero_rule:`
`\g__driver_draw_eor_bool`

The even-odd rule here can be implemented as a simply switch.

```
19049 \cs_new_protected:Npn \__driver_draw_evenodd_rule:
19050     { \bool_gset_true:N \g__driver_draw_eor_bool }
19051 \cs_new_protected:Npn \__driver_draw_nonzero_rule:
19052     { \bool_gset_false:N \g__driver_draw_eor_bool }
19053 \bool_new:N \g__driver_draw_eor_bool
```

(*End definition for* `\__driver_draw_evenodd_rule:` *,* `\__driver_draw_nonzero_rule:` *, and* `\g__driver_-draw_eor_bool`*.*)

`\__driver_draw_closepath:`
`\__driver_draw_stroke:`
`\__driver_draw_closestroke:`
`\__driver_draw_fill:`
`\__driver_draw_fillstroke:`
`\__driver_draw_clip:`
`\__driver_draw_discardpath:`

Converting paths to output is again a case of mapping directly to PDF operations.

```
19054 \cs_new_protected:Npn \__driver_draw_closepath:
19055     { \__driver_draw_literal:n { h } }
19056 \cs_new_protected:Npn \__driver_draw_stroke:
19057     { \__driver_draw_literal:n { S } }
19058 \cs_new_protected:Npn \__driver_draw_closestroke:
19059     { \__driver_draw_literal:n { s } }
19060 \cs_new_protected:Npn \__driver_draw_fill:
19061     {
19062         \__driver_draw_literal:x
19063             { f \bool_if:NT \g__driver_draw_eor_bool * }
19064     }
19065 \cs_new_protected:Npn \__driver_draw_fillstroke:
19066     {
19067         \__driver_draw_literal:x
19068             { B \bool_if:NT \g__driver_draw_eor_bool * }
19069     }
19070 \cs_new_protected:Npn \__driver_draw_clip:
19071     {
19072         \__driver_draw_literal:x
19073             { W \bool_if:NT \g__driver_draw_eor_bool * }
19074     }
19075 \cs_new_protected:Npn \__driver_draw_discardpath:
19076     { \__driver_draw_literal:n { n } }
```

(*End definition for* `\__driver_draw_closepath:` *and others.*)

`\__driver_draw_dash:nn`
`\__driver_draw_dash:n`
`\__driver_draw_linewidth:n`
`\__driver_draw_miterlimit:n`
`\__driver_draw_cap_butt:`
`\__driver_draw_cap_round:`
`\__driver_draw_cap_rectangle:`
`\__driver_draw_join_miter:`
`\__driver_draw_join_round:`
`\__driver_draw_join_bevel:`

Converting paths to output is again a case of mapping directly to PDF operations.

```
19077 \cs_new_protected:Npn \__driver_draw_dash:nn #1#2
19078     {
19079         \__driver_draw_literal:x
19080             {
19081                 [ ~
19082                     \clist_map_function:nN {#1} \__driver_draw_dash:n
19083                 ] ~
19084                 \dim_to_decimal_in_bp:n {#2} ~ d
19085             }
19086     }
19087 \cs_new:Npn \__driver_draw_dash:n #1
19088     { \dim_to_decimal_in_bp:n {#1} ~ }
19089 \cs_new_protected:Npn \__driver_draw_linewidth:n #1
19090     {
19091         \__driver_draw_literal:x
19092             { \dim_to_decimal_in_bp:n {#1} ~ w }
```

```
19093        }
19094    \cs_new_protected:Npn \__driver_draw_miterlimit:n #1
19095      { \__driver_draw_literal:x { \fp_eval:n {#1} ~ M } }
19096    \cs_new_protected:Npn \__driver_draw_cap_butt:
19097      { \__driver_draw_literal:n { 0 ~ J } }
19098    \cs_new_protected:Npn \__driver_draw_cap_round:
19099      { \__driver_draw_literal:n { 1 ~ J } }
19100    \cs_new_protected:Npn \__driver_draw_cap_rectangle:
19101      { \__driver_draw_literal:n { 2 ~ J } }
19102    \cs_new_protected:Npn \__driver_draw_join_miter:
19103      { \__driver_draw_literal:n { 0 ~ j } }
19104    \cs_new_protected:Npn \__driver_draw_join_round:
19105      { \__driver_draw_literal:n { 1 ~ j } }
19106    \cs_new_protected:Npn \__driver_draw_join_bevel:
19107      { \__driver_draw_literal:n { 2 ~ j } }
```

(*End definition for* `\__driver_draw_dash:nn` *and others.*)

Yet more fast conversion, all using the FPU to allow for expressions in numerical input.

`\_driver_draw_color_cmyk:nnnn`
`\_driver_draw_color_cmyk_fill:nnnn`
`\_driver_draw_color_cmyk_stroke:nnnn`
`\_driver_draw_color_cmyk_aux:nnnn`
`\__driver_draw_color_gray:n`
`\_driver_draw_color_gray_fill:n`
`\_driver_draw_color_gray_stroke:n`
`\_driver_draw_color_gray_aux:n`
`\__driver_draw_color_rgb:nnn`
`\_driver_draw_color_rgb_fill:nnn`
`\_driver_draw_color_rgb_stroke:nnn`
`\_driver_draw_color_rgb_aux:nnn`

```
19108    \cs_new_protected:Npn \__driver_draw_color_cmyk:nnnn #1#2#3#4
19109      {
19110        \use:x
19111          {
19112            \__driver_draw_color_cmyk_aux:nnnn
19113              { \fp_eval:n {#1} }
19114              { \fp_eval:n {#2} }
19115              { \fp_eval:n {#3} }
19116              { \fp_eval:n {#4} }
19117          }
19118      }
19119    \cs_new_protected:Npn \__driver_draw_color_cmyk_aux:nnnn #1#2#3#4
19120      {
19121        \__driver_draw_literal:n
19122          { #1 ~ #2 ~ #3 ~ #4 ~ k ~ #1 ~ #2 ~ #3 ~ #4 ~ K }
19123      }
19124    \cs_new_protected:Npn \__driver_draw_color_cmyk_fill:nnnn #1#2#3#4
19125      {
19126        \__driver_draw_literal:x
19127          {
19128            \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
19129            \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
19130            k
19131          }
19132      }
19133    \cs_new_protected:Npn \__driver_draw_color_cmyk_stroke:nnnn #1#2#3#4
19134      {
19135        \__driver_draw_literal:x
19136          {
19137            \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
19138            \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
19139            K
19140          }
19141      }
19142    \cs_new_protected:Npn \__driver_draw_color_gray:n #1
```

```
19143        {
19144          \use:x
19145            { \__driver_draw_color_gray_aux:n { \fp_eval:n {#1} } }
19146        }
19147     \cs_new_protected:Npn \__driver_draw_color_gray_aux:n #1
19148        {
19149          \__driver_draw_literal:n { #1 ~ g ~ #1 ~ G }
19150        }
19151     \cs_new_protected:Npn \__driver_draw_color_gray_fill:n #1
19152        { \__driver_draw_literal:x { \fp_eval:n {#1} ~ g } }
19153     \cs_new_protected:Npn \__driver_draw_color_gray_stroke:n #1
19154        { \__driver_draw_literal:x { \fp_eval:n {#1} ~ G } }
19155     \cs_new_protected:Npn \__driver_draw_color_rgb:nnn #1#2#3
19156        {
19157          \use:x
19158            {
19159              \__driver_draw_color_rgb_aux:nnn
19160                { \fp_eval:n {#1} }
19161                { \fp_eval:n {#2} }
19162                { \fp_eval:n {#3} }
19163            }
19164        }
19165     \cs_new_protected:Npn \__driver_draw_color_rgb_aux:nnn #1#2#3
19166        {
19167          \__driver_draw_literal:n
19168            { #1 ~ #2 ~ #3 ~ rg ~ #1 ~ #2 ~ #3 ~ RG }
19169        }
19170     \cs_new_protected:Npn \__driver_draw_color_rgb_fill:nnn #1#2#3
19171        {
19172          \__driver_draw_literal:x
19173            { \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~ rg }
19174        }
19175     \cs_new_protected:Npn \__driver_draw_color_rgb_stroke:nnn #1#2#3
19176        {
19177          \__driver_draw_literal:x
19178            { \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~ RG }
19179        }
```

(*End definition for* `\__driver_draw_color_cmyk:nnnn` *and others.*)

`\__driver_draw_transformcm:nnnnnn` The first four arguments here are floats (the affine matrix), the last two are a displacement vector. Once again, force evaluation to allow for caching.

```
19180     \cs_new_protected:Npn \__driver_draw_transformcm:nnnnnn #1#2#3#4#5#6
19181        {
19182          \__driver_draw_literal:x
19183            {
19184              \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
19185              \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
19186              \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
19187              cm
19188            }
19189        }
```

(*End definition for* `\__driver_draw_transformcm:nnnnnn.`)

Inserting a TeX box transformed to the requested position and using the current matrix is done using a mixture of TeX and low-level manipulation. The offset can be handled by TeX, so only any rotation/skew/scaling component needs to be done using the matrix operation. As this operation can never be cached, the scope is set directly not using the draw version.

```
19190 \cs_new_protected:Npn \__driver_draw_hbox:Nnnnnnn #1#2#3#4#5#6#7
19191   {
19192     \hbox_set:Nn \l__driver_tmp_box
19193       {
19194         \tex_kern:D \__dim_eval:w #6 \__dim_eval_end:
19195         \__driver_scope_begin:
19196         \__driver_draw_transformcm:nnnnnn {#2} {#3} {#4} {#5}
19197           { 0pt } { 0pt }
19198         \box_move_up:nn {#7} { \box_use:N #1 }
19199         \__driver_scope_end:
19200       }
19201     \box_set_wd:Nn \l__driver_tmp_box { 0pt }
19202     \box_set_ht:Nn \l__driver_tmp_box { 0pt }
19203     \box_set_dp:Nn \l__driver_tmp_box { 0pt }
19204     \box_use:N \l__driver_tmp_box
19205   }
19206 \box_new:N \l__driver_tmp_box
```

(*End definition for* __driver_draw_hbox:Nnnnnnn *and* \l__driver_tmp_box.)

```
19207 ⟨/dvipdfmx | pdfmode | xdvipdfmx⟩
```

37.6 dvips driver

```
19208 ⟨*dvips⟩
```

37.6.1 Basics

__driver_literal:n In the case of dvips there is no build-in saving of the current position, and so some additional PostScript is required to set up the transformation matrix and also to restore it afterwards. Notice the use of the stack to save the current position "up front" and to move back to it at the end of the process.

```
19209 \cs_new_protected:Npn \__driver_literal:n #1
19210   {
19211     \tex_special:D
19212       {
19213         ps:
19214           currentpoint~
19215           currentpoint~translate~
19216           #1 ~
19217           neg~exch~neg~exch~translate
19218       }
19219   }
```

(*End definition for* __driver_literal:n.)

__driver_scope_begin: Scope saving/restoring is done directly with no need to worry about the transformation
__driver_scope_end: matrix. General scoping is only for the graphics stack so the lower-cost gsave/grestore pair are used.

```
19220 \cs_new_protected:Npn \__driver_scope_begin:
19221   { \tex_special:D { ps:gsave } }
19222 \cs_new_protected:Npn \__driver_scope_end:
19223   { \tex_special:D { ps:grestore } }
```

(*End definition for* __driver_scope_begin: *and* __driver_scope_end:.)

37.7 Driver-specific auxiliaries

__driver_absolute_lengths:n   The dvips driver scales all absolute dimensions based on the output resolution selected
and any TeX magnification. Thus for any operation involving absolute lengths there is
a correction to make. This is based on normalscale from special.pro but using the
stack rather than a definition to save the current matrix.

```
19224 \cs_new:Npn \__driver_absolute_lengths:n #1
19225   {
19226     matrix~currentmatrix~
19227     Resolution~72~div~VResolution~72~div~scale~
19228     DVImag~dup~scale~
19229     #1 ~
19230     setmatrix
19231   }
```

(*End definition for* __driver_absolute_lengths:n.)

37.7.1 Box operations

__driver_box_use_clip:N   Much the same idea as for the PDF mode version but with a slightly different syntax for
creating the clip path. To avoid any scaling issues we need the absolute length auxiliary
here.

```
19232 \cs_new_protected:Npn \__driver_box_use_clip:N #1
19233   {
19234     \__driver_scope_begin:
19235     \__driver_literal:n
19236       {
19237         \__driver_absolute_lengths:n
19238           {
19239             0 ~
19240             \dim_to_decimal_in_bp:n { \box_dp:N #1 } ~
19241             \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
19242             \dim_to_decimal_in_bp:n { -\box_ht:N #1 - \box_dp:N #1 } ~
19243             rectclip
19244           }
19245       }
19246     \hbox_overlap_right:n { \box_use:N #1 }
19247     \__driver_scope_end:
19248     \skip_horizontal:n { \box_wd:N #1 }
19249   }
```

(*End definition for* __driver_box_use_clip:N.)

__driver_box_use_rotate:Nn   Rotating using dvips does not require that the box dimensions are altered and has a
very convenient built-in operation. Zero rotation must be written as 0 not -0 so there is
a quick test.

```
19250 \cs_new_protected:Npn \__driver_box_use_rotate:Nn #1#2
```

```
19251   {
19252     \__driver_scope_begin:
19253     \__driver_literal:n
19254       {
19255         \fp_compare:nNnTF {#2} = \c_zero_fp
19256           { 0 }
19257           { \fp_eval:n { round ( -#2 , 5 ) } } } ~
19258         rotate
19259       }
19260     \box_use:N #1
19261     \__driver_scope_end:
19262   }
19263 % \end{macro}
19264 %
19265 % \begin{macro}{\__driver_box_use_scale:Nnn}
19266 %   The \texttt{dvips} driver once again has a dedicated operation we can
19267 %   use here.
19268 %     \begin{macrocode}
19269 \cs_new_protected:Npn \__driver_box_use_scale:Nnn #1#2#3
19270   {
19271     \__driver_scope_begin:
19272     \__driver_literal:n
19273       {
19274         \fp_eval:n { round ( #2 , 5 ) } ~
19275         \fp_eval:n { round ( #3 , 5 ) } ~
19276         scale
19277       }
19278     \hbox_overlap_right:n { \box_use:N #1 }
19279     \__driver_scope_end:
19280   }
```

(*End definition for* __driver_box_use_rotate:Nn.)

37.7.2   Color

\l__driver_current_color_tl   The current color in driver-dependent format.

```
19281 \tl_new:N \l__driver_current_color_tl
19282 \tl_set:Nn \l__driver_current_color_tl { gray~0 }
19283 ⟨*package⟩
19284 \AtBeginDocument
19285   {
19286     \@ifpackageloaded { color }
19287       { \tl_set:Nn \l__driver_current_color_tl { \current@color } }
19288       { }
19289   }
19290 ⟨/package⟩
```

(*End definition for* \l__driver_current_color_tl.)

__driver_color_ensure_current: Directly set the color using the specials: no optimisation here.
__driver_color_reset:
```
19291 \cs_new_protected:Npn \__driver_color_ensure_current:
19292   {
19293     \tex_special:D { color~push~\l__driver_current_color_tl }
19294     \group_insert_after:N \__driver_color_reset:
```

```
19295      }
19296  \cs_new_protected:Npn \__driver_color_reset:
19297      { \tex_special:D { color~pop } }
```

(*End definition for* `\__driver_color_ensure_current:` *and* `\__driver_color_reset:`.)

37.8   Drawing

`\__driver_draw_literal:n`
`\__driver_draw_literal:x`

Literals with no positioning (using `ps:` each one is positioned but cut of from everything else, so no good for the stepwise approach needed here).

```
19298  \cs_new_protected:Npn \__driver_draw_literal:n #1
19299      { \tex_special:D { ps:: ~ #1 } }
19300  \cs_generate_variant:Nn \__driver_draw_literal:n { x }
```

(*End definition for* `\__driver_draw_literal:n`.)

`\__driver_draw_begin:`
`\__driver_draw_end:`

The `ps::[begin]` special here deals with positioning but allows us to continue on to a matching `ps::[end]`: contrast with `ps:`, which positions but where we can't split material between separate calls. The `@beginspecial`/`@endspecial` pair are from `special.pro` and correct the scale and *y*-axis direction. The reference point at the start of the box is saved (as `l3x`/`l3y`) as it is needed when inserting various items.

```
19301  \cs_new_protected:Npn \__driver_draw_begin:
19302      {
19303          \tex_special:D { ps::[begin] }
19304          \tex_special:D { ps::~save }
19305          \tex_special:D { ps::~/l3x~currentpoint~/l3y~exch~def~def }
19306          \tex_special:D { ps::~@beginspecial }
19307      }
19308  \cs_new_protected:Npn \__driver_draw_end:
19309      {
19310          \tex_special:D { ps::~@endspecial }
19311          \tex_special:D { ps::~restore }
19312          \tex_special:D { ps::[end] }
19313      }
```

(*End definition for* `\__driver_draw_begin:` *and* `\__driver_draw_end:`.)

`\__driver_draw_scope_begin:`
`\__driver_draw_scope_end:`

Scope here may need to contain saved definitions, so the entire memory rather than just the graphic state has to be sent to the stack.

```
19314  \cs_new_protected:Npn \__driver_draw_scope_begin:
19315      { \__driver_draw_literal:n { save } }
19316  \cs_new_protected:Npn \__driver_draw_scope_end:
19317      { \__driver_draw_literal:n { restore } }
```

(*End definition for* `\__driver_draw_scope_begin:` *and* `\__driver_draw_scope_end:`.)

`\__driver_draw_moveto:nn`
`\__driver_draw_lineto:nn`
`\__driver_draw_rectangle:nnnn`
`\__driver_draw_curveto:nnnnnn`

Path creation operations mainly resolve directly to PostScript primitive steps, with only the need to convert to `bp`. Notice that x-type expansion is included here to ensure that any variable values are forced to literals before any possible caching. There is no native rectangular path command (without also clipping, filling or stroking), so that task is done using a small amount of PostScript.

```
19318  \cs_new_protected:Npn \__driver_draw_moveto:nn #1#2
19319      {
19320          \__driver_draw_literal:x
```

791

```
19321          { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ moveto }
19322      }
19323  \cs_new_protected:Npn \__driver_draw_lineto:nn #1#2
19324    {
19325      \__driver_draw_literal:x
19326        { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ lineto }
19327      }
19328  \cs_new_protected:Npn \__driver_draw_rectangle:nnnn #1#2#3#4
19329    {
19330      \__driver_draw_literal:x
19331        {
19332          \dim_to_decimal_in_bp:n {#4} ~ \dim_to_decimal_in_bp:n {#3} ~
19333          \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
19334          moveto~dup~0~rlineto~exch~0~exch~rlineto~neg~0~rlineto~clospath
19335        }
19336      }
19337  \cs_new_protected:Npn \__driver_draw_curveto:nnnnnn #1#2#3#4#5#6
19338    {
19339      \__driver_draw_literal:x
19340        {
19341          \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
19342          \dim_to_decimal_in_bp:n {#3} ~ \dim_to_decimal_in_bp:n {#4} ~
19343          \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
19344          curveto
19345        }
19346      }
```

(*End definition for* __driver_draw_moveto:nn *and others.*)

__driver_draw_evenodd_rule: The even-odd rule here can be implemented as a simply switch.
__driver_draw_nonzero_rule:
\g__driver_draw_eor_bool
```
19347  \cs_new_protected:Npn \__driver_draw_evenodd_rule:
19348    { \bool_gset_true:N \g__driver_draw_eor_bool }
19349  \cs_new_protected:Npn \__driver_draw_nonzero_rule:
19350    { \bool_gset_false:N \g__driver_draw_eor_bool }
19351  \bool_new:N \g__driver_draw_eor_bool
```

(*End definition for* __driver_draw_evenodd_rule: , __driver_draw_nonzero_rule: , *and* \g__driver_-
draw_eor_bool.)

__driver_draw_closepath: Unlike PDF, PostScript doesn't track separate colors for strokes and other elements. It is
__driver_draw_stroke: also desirable to have the clip keyword after a stroke or fill. To achieve those outcomes,
__driver_draw_closestroke: there is some work to do. For color, if a stroke or fill color is defined it is used for the
__driver_draw_fill: relevant operation, with a graphic scope inserted as required. That does mean that once
__driver_draw_fillstroke: such a color is set all further uses inside the same scope have to use scoping: see also
__driver_draw_clip: the color set up functions. For clipping, the required ordering is achieved using a TEX
__driver_draw_discardpath: switch. All of the operations end with a new path instruction as they do not terminate
\g__driver_draw_clip_bool   (again in contrast to PDF).

```
19352  \cs_new_protected:Npn \__driver_draw_closepath:
19353    { \__driver_draw_literal:n { closepath } }
19354  \cs_new_protected:Npn \__driver_draw_stroke:
19355    {
19356      \__driver_draw_literal:n { currentdict~/l3sc~known~{gsave~l3sc}~if }
19357      \__driver_draw_literal:n { stroke }
19358      \__driver_draw_literal:n { currentdict~/l3sc~known~{grestore}~if }
```

```
19359        \bool_if:NT \g__driver_draw_clip_bool
19360          {
19361            \__driver_draw_literal:x
19362              {
19363                \bool_if:NT \g__driver_draw_eor_bool { eo }
19364                clip
19365              }
19366          }
19367        \__driver_draw_literal:n { newpath }
19368        \bool_gset_false:N \g__driver_draw_clip_bool
19369      }
19370    \cs_new_protected:Npn \__driver_draw_closestroke:
19371      {
19372        \__driver_draw_closepath:
19373        \__driver_draw_stroke:
19374      }
19375    \cs_new_protected:Npn \__driver_draw_fill:
19376      {
19377        \__driver_draw_literal:n { currentdict~/l3fc~known~{gsave~l3fc}~if }
19378        \__driver_draw_literal:x
19379          {
19380            \bool_if:NT \g__driver_draw_eor_bool { eo }
19381            fill
19382          }
19383        \__driver_draw_literal:n { currentdict~/l3fc~known~{grestore}~if }
19384        \bool_if:NT \g__driver_draw_clip_bool
19385          {
19386            \__driver_draw_literal:x
19387              {
19388                \bool_if:NT \g__driver_draw_eor_bool { eo }
19389                clip
19390              }
19391          }
19392        \__driver_draw_literal:n { newpath }
19393        \bool_gset_false:N \g__driver_draw_clip_bool
19394      }
19395    \cs_new_protected:Npn \__driver_draw_fillstroke:
19396      {
19397        \__driver_draw_literal:n { currentdict~/l3fc~known~{gsave~l3fc}~if }
19398        \__driver_draw_literal:x
19399          {
19400            \bool_if:NT \g__driver_draw_eor_bool { eo }
19401            fill
19402          }
19403        \__driver_draw_literal:n { currentdict~/l3fc~known~{grestore}~if }
19404        \__driver_draw_literal:n { currentdict~/l3sc~known~{gsave~l3sc}~if }
19405        \__driver_draw_literal:n { stroke }
19406        \__driver_draw_literal:n { currentdict~/l3sc~known~{grestore}~if }
19407        \bool_if:NT \g__driver_draw_clip_bool
19408          {
19409            \__driver_draw_literal:x
19410              {
19411                \bool_if:NT \g__driver_draw_eor_bool { eo }
19412                clip
```

```
19413                }
19414              }
19415           \__driver_draw_literal:n { newpath }
19416           \bool_gset_false:N \g__driver_draw_clip_bool
19417        }
19418     \cs_new_protected:Npn \__driver_draw_clip:
19419        { \bool_gset_true:N \g__driver_draw_clip_bool }
19420     \bool_new:N \g__driver_draw_clip_bool
19421     \cs_new_protected:Npn \__driver_draw_discardpath:
19422        {
19423           \bool_if:NT \g__driver_draw_clip_bool
19424             {
19425                \__driver_draw_literal:x
19426                   {
19427                      \bool_if:NT \g__driver_draw_eor_bool { eo }
19428                      clip
19429                   }
19430             }
19431           \__driver_draw_literal:n { newpath }
19432           \bool_gset_false:N \g__driver_draw_clip_bool
19433        }
```

(*End definition for* `\__driver_draw_closepath:` *and others.*)

Converting paths to output is again a case of mapping directly to PostScript operations.

```
19434     \cs_new_protected:Npn \__driver_draw_dash:nn #1#2
19435        {
19436           \__driver_draw_literal:x
19437             {
19438                [ ~
19439                   \clist_map_function:nN {#1} \__driver_draw_dash:n
19440                ] ~
19441                \dim_to_decimal_in_bp:n {#2} ~ setdash
19442             }
19443        }
19444     \cs_new:Npn \__driver_draw_dash:n #1
19445        { \dim_to_decimal_in_bp:n {#1} ~ }
19446     \cs_new_protected:Npn \__driver_draw_linewidth:n #1
19447        {
19448           \__driver_draw_literal:x
19449             { \dim_to_decimal_in_bp:n {#1} ~ setlinewidth }
19450        }
19451     \cs_new_protected:Npn \__driver_draw_miterlimit:n #1
19452        { \__driver_draw_literal:x { \fp_eval:n {#1} ~ setmiterlimit } }
19453     \cs_new_protected:Npn \__driver_draw_cap_butt:
19454        { \__driver_draw_literal:n { 0 ~ setlinecap } }
19455     \cs_new_protected:Npn \__driver_draw_cap_round:
19456        { \__driver_draw_literal:n { 1 ~ setlinecap } }
19457     \cs_new_protected:Npn \__driver_draw_cap_rectangle:
19458        { \__driver_draw_literal:n { 2 ~ setlinecap } }
19459     \cs_new_protected:Npn \__driver_draw_join_miter:
19460        { \__driver_draw_literal:n { 0 ~ setlinejoin } }
19461     \cs_new_protected:Npn \__driver_draw_join_round:
19462        { \__driver_draw_literal:n { 1 ~ setlinejoin } }
```

```
19463  \cs_new_protected:Npn \__driver_draw_join_bevel:
19464    { \__driver_draw_literal:n { 2 ~ setlinejoin } }
```

(*End definition for* __driver_draw_dash:nn *and others.*)

To allow color to be defined for strokes and fills separately and to respect scoping, the data needs to be stored at the PostScript level. We cannot undefine (local) fill/stroke colors once set up but we can set them blank to improve performance slightly.

```
19465  \cs_new_protected:Npn \__driver_draw_color_reset:
19466    {
19467      \__driver_draw_literal:n { currentdic~/l3fc~known~{ /l3fc~ { } ~def }~if }
19468      \__driver_draw_literal:n { currentdic~/l3sc~known~{ /l3sc~ { } ~def }~if }
19469    }
19470  \cs_new_protected:Npn \__driver_draw_color_cmyk:nnnn #1#2#3#4
19471    {
19472      \__driver_draw_literal:x
19473        {
19474          \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
19475          \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
19476          setcmykcolor ~
19477        }
19478      \__driver_draw_color_reset:
19479    }
19480  \cs_new_protected:Npn \__driver_draw_color_cmyk_fill:nnnn #1#2#3#4
19481    {
19482      \__driver_draw_literal:x
19483        {
19484          /l3fc ~
19485            {
19486              \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
19487              \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
19488              setcmykcolor
19489            } ~
19490          def
19491        }
19492    }
19493  \cs_new_protected:Npn \__driver_draw_color_cmyk_stroke:nnnn #1#2#3#4
19494    {
19495      \__driver_draw_literal:x
19496        {
19497          /l3sc ~
19498            {
19499              \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
19500              \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
19501              setcmykcolor
19502            } ~
19503          def
19504        }
19505    }
19506  \cs_new_protected:Npn \__driver_draw_color_gray:n #1
19507    {
19508      \__driver_draw_literal:x { fp_eval:n {#1} ~ setgray  }
19509      \__driver_draw_color_reset:
19510    }
```

795

```
19511  \cs_new_protected:Npn \__driver_draw_color_gray_fill:n #1
19512    { \__driver_draw_literal:x { /l3fc ~ { \fp_eval:n {#1} ~ setgray } ~ def } }
19513  \cs_new_protected:Npn \__driver_draw_color_gray_stroke:n #1
19514    { \__driver_draw_literal:x { /l3sc ~ { \fp_eval:n {#1} ~ setgray } ~ def } }
19515  \cs_new_protected:Npn \__driver_draw_color_rgb:nnn #1#2#3
19516    {
19517      \__driver_draw_literal:x
19518        {
19519          \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~
19520          setrgbcolor
19521        }
19522      \__driver_draw_color_reset:
19523    }
19524  \cs_new_protected:Npn \__driver_draw_color_rgb_fill:nnn #1#2#3
19525    {
19526      \__driver_draw_literal:x
19527        {
19528          /l3fc ~
19529            {
19530              \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~
19531              setrgbcolor
19532            } ~
19533          def
19534        }
19535    }
19536  \cs_new_protected:Npn \__driver_draw_color_rgb_stroke:nnn #1#2#3
19537    {
19538      \__driver_draw_literal:x
19539        {
19540          /l3sc ~
19541            {
19542              \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~
19543              setrgbcolor
19544            } ~
19545          def
19546        }
19547    }
```

(*End definition for* __driver_draw_color_reset: *and others.*)

__driver_draw_transformcm:nnnnnn   The first four arguments here are floats (the affine matrix), the last two are a displacement
vector. Once again, force evaluation to allow for caching.

```
19548  \cs_new_protected:Npn \__driver_draw_transformcm:nnnnnn #1#2#3#4#5#6
19549    {
19550      \__driver_draw_literal:x
19551        {
19552          [
19553            \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
19554            \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
19555            \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
19556          ] ~
19557          concat
19558        }
19559    }
```

(*End definition for* `\__driver_draw_transformcm:nnnnnn`.)

`\__driver_draw_hbox:Nnnnnnn` Inside a picture @beginspecial/@endspecial are active, which is normally a good thing but means that the position and scaling will be off if the box is inserted directly. Instead, we need to reverse the effect of the (normally desirable) shift/scaling within the box. That requires knowing where the reference point for the drawing is: saved as `l3x`/`l3y` at the start of the picture. Transformation here is relative to the drawing origin so has to be done purely in driver code not using TEX offsets.

```
19560 \cs_new_protected:Npn \__driver_draw_hbox:Nnnnnnn #1#2#3#4#5#6#7
19561   {
19562     \__driver_scope_begin:
19563     \tex_special:D { ps::[end] }
19564     \__driver_draw_transformcm:nnnnnn {#2} {#3} {#4} {#5} {#6} {#7}
19565     \tex_special:D { ps::~72~Resolution~div~72~VResolution~div~neg~scale }
19566     \tex_special:D { ps::~magscale~{1~DVImag~div~dup~scale}~if }
19567     \tex_special:D { ps::~l3x~neg~l3y~neg~translate }
19568     \group_begin:
19569       \box_set_wd:Nn #1 { 0pt }
19570       \box_set_ht:Nn #1 { 0pt }
19571       \box_set_dp:Nn #1 { 0pt }
19572       \box_use:N #1
19573     \group_end:
19574     \tex_special:D { ps::[begin] }
19575     \__driver_scope_end:
19576   }
```

(*End definition for* `\__driver_draw_hbox:Nnnnnnn`.)

```
19577 ⟨/dvips⟩
```

37.9 dvisvgm driver

```
19578 ⟨*dvisvgm⟩
```

37.9.1 Basics

`\__driver_literal:n` Unlike the other drivers, the requirements for making SVG files mean that we can't conveniently transform all operations to the current point. That makes life a bit more tricky later as that needs to be accounted for. A new line is added after each call to help to keep the output readable for debugging.

```
19579 \cs_new_protected:Npn \__driver_literal:n #1
19580   { \tex_special:D { dvisvgm:raw~ #1 { ?nl } } }
```

(*End definition for* `\__driver_literal:n`.)

`\__driver_scope_begin:`
`\__driver_scope_end:` A scope in SVG terms is slightly different to the other drivers as operations have to be "tied" to these not simply inside them.

```
19581 \cs_new_protected:Npn \__driver_scope_begin:
19582   { \__driver_literal:n { <g> } }
19583 \cs_new_protected:Npn \__driver_scope_end:
19584   { \__driver_literal:n { </g> } }
```

(*End definition for* `\__driver_scope_begin:` *and* `\__driver_scope_end:`.)

37.10 Driver-specific auxiliaries

__driver_scope_begin:n

In SVG transformations, clips and so on are attached directly to scopes so we need a way or allowing for that. This is rather more useful that __driver_scope_begin: as a result. No assumptions are made about the nature of the scoped operation(s).

```
19585 \cs_new_protected:Npn \__driver_scope_begin:n #1
19586   { \__driver_literal:n { <g~ #1 > } }
```

(*End definition for* __driver_scope_begin:n.)

37.10.1 Box operations

__driver_box_use_clip:N
\g__driver_clip_path_int

Clipping in SVG is more involved than with other drivers. The first issue is that the clipping path must be defined separately from where it is used, so we need to track how many paths have applied. The naming here uses `l3cp` as the namespace with a number following. Rather than use a rectangular operation, we define the path manually as this allows it to have a depth: easier than the alternative approach of shifting content up and down using scopes to allow for the depth of the TeX box and keep the reference point the same!

```
19587 \cs_new_protected:Npn \__driver_box_use_clip:N #1
19588   {
19589     \int_gincr:N \g__driver_clip_path_int
19590     \__driver_literal:n
19591       { < clipPath~id = " l3cp \int_use:N \g__driver_clip_path_int " > }
19592     \__driver_literal:n
19593       {
19594         <
19595           path ~ d =
19596             "
19597               M ~ 0 ~
19598                 \dim_to_decimal:n { -\box_dp:N #1 } ~
19599               L ~ \dim_to_decimal:n { \box_wd:N #1 } ~
19600                 \dim_to_decimal:n { -\box_dp:N #1 } ~
19601               L ~ \dim_to_decimal:n { \box_wd:N #1 }  ~
19602                 \dim_to_decimal:n { \box_ht:N #1 + \box_dp:N #1 } ~
19603               L ~ 0 ~
19604                 \dim_to_decimal:n { \box_ht:N #1 + \box_dp:N #1 } ~
19605               Z
19606             "
19607         />
19608       }
19609     \__driver_literal:n
19610       { < /clipPath > }
```

In general the SVG set up does not try to transform coordinates to the current point. For clipping we need to do that, so have a transformation here to get us to the right place, and a matching one just before the TeX box is inserted to get things back on track. The clip path needs to come between those two such that if lines up with the current point, as does the TeX box.

```
19611       \__driver_scope_begin:n
19612         {
19613           transform =
19614             "
```

```
19615              translate ( { ?x } , { ?y } ) ~
19616              scale ( 1 , -1 )
19617            "
19618        }
19619      \__driver_scope_begin:n
19620        {
19621          clip-path = "url ( \c_hash_str l3cp \int_use:N \g__driver_clip_path_int ) "
19622        }
19623      \__driver_scope_begin:n
19624        {
19625          transform =
19626            "
19627              scale ( -1 , 1 ) ~
19628              translate ( { ?x } , { ?y } ) ~
19629              scale ( -1 , -1 )
19630            "
19631        }
19632      \box_use:N #1
19633      \__driver_scope_end:
19634      \__driver_scope_end:
19635      \__driver_scope_end:
19636 %      \skip_horizontal:n { \box_wd:N #1 }
19637    }
19638 \int_new:N \g__driver_clip_path_int
```

(*End definition for* __driver_box_use_clip:N *and* \g__driver_clip_path_int.)

__driver_box_use_rotate:Nn   Rotation has a dedicated operation which includes a centre-of-rotation optional pair.
That can be picked up from the driver syntax, so there is no need to worry about the
transformation matrix.

```
19639 \cs_new_protected:Npn \__driver_box_use_rotate:Nn #1#2
19640    {
19641      \__driver_scope_begin:n
19642        {
19643          transform =
19644            "
19645              rotate
19646              ( \fp_eval:n { round ( -#2 , 5 ) } , ~ { ?x } , ~ { ?y } )
19647            "
19648        }
19649      \box_use:N #1
19650      \__driver_scope_end:
19651    }
```

(*End definition for* __driver_box_use_rotate:Nn.)

__driver_box_use_scale:Nnn   In contrast to rotation, we have to account for the current position in this case. That
is done using a couple of translations in addition to the scaling (which is therefore done
backward with a flip).

```
19652 \cs_new_protected:Npn \__driver_box_use_scale:Nnn #1#2#3
19653    {
19654      \__driver_scope_begin:n
19655        {
19656          transform =
```

```
19657                    "
19658                        translate ( { ?x } , { ?y } ) ~
19659                        scale
19660                          (
19661                             \fp_eval:n { round ( -#2 , 5 ) } ,
19662                             \fp_eval:n { round ( -#3 , 5 ) }
19663                          ) ~
19664                        translate ( { ?x } , { ?y } ) ~
19665                        scale ( -1 )
19666                    "
19667              }
19668          \hbox_overlap_right:n { \box_use:N #1 }
19669          \__driver_scope_end:
19670      }
```

(*End definition for* __driver_box_use_scale:Nnn.)

37.10.2 Color

\l__driver_current_color_tl   The current color in driver-dependent format: the same as for dvips.

```
19671 \tl_new:N \l__driver_current_color_tl
19672 \tl_set:Nn \l__driver_current_color_tl { gray~0 }
19673 ⟨*package⟩
19674 \AtBeginDocument
19675     {
19676        \@ifpackageloaded { color }
19677          { \tl_set:Nn \l__driver_current_color_tl { \current@color } }
19678          { }
19679     }
19680 ⟨/package⟩
```

(*End definition for* \l__driver_current_color_tl.)

__driver_color_ensure_current: Directly set the color: same as dvips.
__driver_color_reset:

```
19681 \cs_new_protected:Npn \__driver_color_ensure_current:
19682     {
19683        \tex_special:D { color~push~\l__driver_current_color_tl }
19684        \group_insert_after:N \__driver_color_reset:
19685     }
19686 \cs_new_protected:Npn \__driver_color_reset:
19687     { \tex_special:D { color~pop } }
```

(*End definition for* __driver_color_ensure_current: *and* __driver_color_reset:.)

37.11 Drawing

__driver_draw_literal:n   The same as the more general literal call.
__driver_draw_literal:x

```
19688 \cs_new_eq:NN \__driver_draw_literal:n \__driver_literal:n
19689 \cs_generate_variant:Nn \__driver_draw_literal:n { x }
```

(*End definition for* __driver_draw_literal:n.)

$\_\_driver\_draw\_begin:$
$\_\_driver\_draw\_end:$

A drawing needs to be set up such that the co-ordinate system is translated. That is done inside a scope, which as described below

```
19690 \cs_new_protected:Npn \__driver_draw_begin:
19691   {
19692     \__driver_draw_scope_begin:
19693     \__driver_draw_scope:n { transform="translate({?x},{?y})~scale(1,-1)" }
19694   }
19695 \cs_new_protected:Npn \__driver_draw_end:
19696   { \__driver_draw_scope_end: }
```

(*End definition for* $\_\_driver\_draw\_begin:$ *and* $\_\_driver\_draw\_end:$.)

$\_\_driver\_draw\_scope\_begin:$
$\_\_driver\_draw\_scope\_end:$
$\_\_driver\_draw\_scope:n$
$\_\_driver\_draw\_scope:x$
$\g\_\_driver\_draw\_scope\_int$
$\l\_\_driver\_draw\_scope\_int$

Several settings that with other drivers are "stand alone" have to be given as part of a scope in SVG. As a result, there is a need to provide a mechanism to automatically close these extra scopes. That is done using a dedicated function and a pair of tracking variables. Within each graphics scope we use a global variable to do the work, with a group used to save the value between scopes. The result is that no direct action is needed when creating a scope.

```
19697 \cs_new_protected:Npn \__driver_draw_scope_begin:
19698   {
19699     \int_set_eq:NN
19700       \l__driver_draw_scope_int
19701       \g__driver_draw_scope_int
19702     \group_begin:
19703       \int_gset:Nn \g__driver_draw_scope_int { 0 }
19704   }
19705 \cs_new_protected:Npn \__driver_draw_scope_end:
19706   {
19707     \prg_replicate:nn
19708       { \g__driver_draw_scope_int }
19709       { \__driver_draw_literal:n { </g> } }
19710     \group_end:
19711     \int_gset_eq:NN
19712       \g__driver_draw_scope_int
19713       \l__driver_draw_scope_int
19714   }
19715 \cs_new_protected:Npn \__driver_draw_scope:n #1
19716   {
19717     \__driver_draw_literal:n { <g~ #1 > }
19718     \int_gincr:N \g__driver_draw_scope_int
19719   }
19720 \cs_generate_variant:Nn \__driver_draw_scope:n { x }
19721 \int_new:N \g__driver_draw_scope_int
19722 \int_new:N \l__driver_draw_scope_int
```

(*End definition for* $\_\_driver\_draw\_scope\_begin:$ *and others.*)

$\_\_driver\_draw\_moveto:nn$
$\_\_driver\_draw\_lineto:nn$
$\_\_driver\_draw\_rectangle:nnnn$
$\_\_driver\_draw\_curveto:nnnnnn$
$\_\_driver\_draw\_add\_to\_path:n$
$\g\_\_driver\_draw\_path\_tl$

Once again, some work is needed to get path constructs correct. Rather then write the values as they are given, the entire path needs to be collected up before being output in one go. For that we use a dedicated storage routine, which will add spaces as required. Since paths should be fully expanded there is no need to worry about the internal x-type expansion.

```
19723 \cs_new_protected:Npn \__driver_draw_moveto:nn #1#2
```

```
19724        {
19725          \__driver_draw_add_to_path:n
19726            { M ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} }
19727        }
19728      \cs_new_protected:Npn \__driver_draw_lineto:nn #1#2
19729        {
19730          \__driver_draw_add_to_path:n
19731            { L ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} }
19732        }
19733      \cs_new_protected:Npn \__driver_draw_rectangle:nnnn #1#2#3#4
19734        {
19735          \__driver_draw_add_to_path:n
19736            {
19737              M ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2}
19738              h ~ \dim_to_decimal:n {#3} ~
19739              v ~ \dim_to_decimal:n {#4} ~
19740              h ~ \dim_to_decimal:n { -#3 } ~
19741              Z
19742            }
19743        }
19744      \cs_new_protected:Npn \__driver_draw_curveto:nnnnnn #1#2#3#4#5#6
19745        {
19746          \__driver_draw_add_to_path:n
19747            {
19748              C ~
19749              \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} ~
19750              \dim_to_decimal:n {#3} ~ \dim_to_decimal:n {#4} ~
19751              \dim_to_decimal:n {#5} ~ \dim_to_decimal:n {#6}
19752            }
19753        }
19754      \cs_new_protected:Npn \__driver_draw_add_to_path:n #1
19755        {
19756          \tl_gset:Nx \g__driver_draw_path_tl
19757            {
19758              \g__driver_draw_path_tl
19759              \tl_if_empty:NF \g__driver_draw_path_tl { \c_space_tl }
19760              #1
19761            }
19762        }
19763      \tl_new:N \g__driver_draw_path_tl
```

(*End definition for* __driver_draw_moveto:nn *and others.*)

__driver_draw_evenodd_rule: The fill rules here have to be handled as scopes.
__driver_draw_nonzero_rule:

```
19764      \cs_new_protected:Npn \__driver_draw_evenodd_rule:
19765        { \__driver_draw_scope:n { fill-rule="evenodd" } }
19766      \cs_new_protected:Npn \__driver_draw_nonzero_rule:
19767        { \__driver_draw_scope:n { fill-rule="nonzero" } }
```

(*End definition for* __driver_draw_evenodd_rule: *and* __driver_draw_nonzero_rule:.)

__driver_draw_path:n   Setting fill and stroke effects and doing clipping all has to be done using scopes. This
__driver_draw_closepath: means setting up the various requirements in a shared auxiliary which deals with the
__driver_draw_stroke: bits and pieces. Clipping paths are reused for path drawing: not essential but avoids
__driver_draw_closestroke:
__driver_draw_fill:
__driver_draw_fillstroke:
__driver_draw_clip:
__driver_draw_discardpath:
\g__driver_draw_clip_bool
\g__driver_draw_path_int

802

constructing them twice. Discarding a path needs a separate function as it's not quite
the same.

```
19768 \cs_new_protected:Npn \__driver_draw_closepath:
19769   { \__driver_draw_add_to_path:n { Z } }
19770 \cs_new_protected:Npn \__driver_draw_path:n #1
19771   {
19772     \bool_if:NTF \g__driver_draw_clip_bool
19773       {
19774         \int_gincr:N \g__driver_clip_path_int
19775         \__driver_draw_literal:x
19776           {
19777             < clipPath~id = " l3cp \int_use:N \g__driver_clip_path_int " >
19778               { ?nl }
19779             <path~d=" \g__driver_draw_path_tl "/> { ?nl }
19780             < /clipPath > { ? nl }
19781             <
19782               use~xlink:href =
19783                 "\c_hash_str l3path \int_use:N \g__driver_path_int " ~
19784                 #1
19785             />
19786           }
19787         \__driver_draw_scope:x
19788           {
19789             clip-path =
19790               "url( \c_hash_str l3cp \int_use:N \g__driver_clip_path_int)"
19791           }
19792       }
19793       {
19794         \__driver_draw_literal:x
19795           { <path ~ d=" \g__driver_draw_path_tl " ~ #1 /> }
19796       }
19797     \tl_gclear:N \g__driver_draw_path_tl
19798     \bool_gset_false:N \g__driver_draw_clip_bool
19799   }
19800 \int_new:N \g__driver_path_int
19801 \cs_new_protected:Npn \__driver_draw_stroke:
19802   { \__driver_draw_path:n { style="fill:none" } }
19803 \cs_new_protected:Npn \__driver_draw_closestroke:
19804   {
19805     \__driver_draw_closepath:
19806     \__driver_draw_stroke:
19807   }
19808 \cs_new_protected:Npn \__driver_draw_fill:
19809   { \__driver_draw_path:n { style="stroke:none" } }
19810 \cs_new_protected:Npn \__driver_draw_fillstroke:
19811   { \__driver_draw_path:n { } }
19812 \cs_new_protected:Npn \__driver_draw_clip:
19813   { \bool_gset_true:N \g__driver_draw_clip_bool }
19814 \bool_new:N \g__driver_draw_clip_bool
19815 \cs_new_protected:Npn \__driver_draw_discardpath:
19816   {
19817     \bool_if:NT \g__driver_draw_clip_bool
19818       {
19819         \int_gincr:N \g__driver_clip_path_int
```

```
19820          \__driver_draw_literal:x
19821            {
19822              < clipPath~id = " l3cp \int_use:N \g__driver_clip_path_int " >
19823                { ?nl }
19824              <path~d=" \g__driver_draw_path_tl "/> { ?nl }
19825              < /clipPath >
19826            }
19827          \__driver_draw_scope:x
19828            {
19829              clip-path =
19830                "url( \c_hash_str l3cp \int_use:N \g__driver_clip_path_int)"
19831            }
19832        }
19833      \tl_gclear:N \g__driver_draw_path_tl
19834      \bool_gset_false:N \g__driver_draw_clip_bool
19835    }
```

(*End definition for* __driver_draw_path:n *and others.*)

__driver_draw_dash:nn All of these ideas are properties of scopes in SVG. The only slight complexity is converting
__driver_draw_dash:n   the dash array properly (doing any required maths).
__driver_draw_dash_aux:nn
__driver_draw_linewidth:n
__driver_draw_miterlimit:n
__driver_draw_cap_butt:
__driver_draw_cap_round:
__driver_draw_cap_rectangle:
__driver_draw_join_miter:
__driver_draw_join_round:
__driver_draw_join_bevel:

```
19836  \cs_new_protected:Npn \__driver_draw_dash:nn #1#2
19837    {
19838      \use:x
19839        {
19840          \__driver_draw_dash_aux:nn
19841            { \clist_map_function:nn {#1} \__driver_draw_dash:n }
19842            { \dim_to_decimal:n {#2} }
19843        }
19844    }
19845  \cs_new:Npn \__driver_draw_dash:n #1
19846    { , \dim_to_decimal_in_bp:n {#1} }
19847  \cs_new_protected:Npn \__driver_draw_dash_aux:nn #1#2
19848    {
19849      \__driver_draw_scope:x
19850        {
19851          stroke-dasharray =
19852            "
19853              \tl_if_empty:oTF { \use_none:n #1 }
19854                { none }
19855                { \use_none:n #1 }
19856            " ~
19857          stroke-offset=" #2 "
19858        }
19859    }
19860  \cs_new_protected:Npn \__driver_draw_linewidth:n #1
19861    { \__driver_draw_scope:x { stroke-width=" \dim_to_decimal:n {#1} " } }
19862  \cs_new_protected:Npn \__driver_draw_miterlimit:n #1
19863    { \__driver_draw_scope:x { stroke-miterlimit=" \fp_eval:n {#1} " } }
19864  \cs_new_protected:Npn \__driver_draw_cap_butt:
19865    { \__driver_draw_scope:n { stroke-linecap="butt" } }
19866  \cs_new_protected:Npn \__driver_draw_cap_round:
19867    { \__driver_draw_scope:n { stroke-linecap="round" } }
19868  \cs_new_protected:Npn \__driver_draw_cap_rectangle:
```

```
19869      { \__driver_draw_scope:n { stroke-linecap="square" } }
19870    \cs_new_protected:Npn \__driver_draw_join_miter:
19871      { \__driver_draw_scope:n { stroke-linejoin="miter" } }
19872    \cs_new_protected:Npn \__driver_draw_join_round:
19873      { \__driver_draw_scope:n { stroke-linejoin="round" } }
19874    \cs_new_protected:Npn \__driver_draw_join_bevel:
19875      { \__driver_draw_scope:n { stroke-linejoin="bevel" } }
```

(*End definition for* __driver_draw_dash:nn *and others.*)

SVG only works with RGB colors, so there is some conversion to do. The values also need to be given as percentages, which means a little more maths.

```
19876    \cs_new_protected:Npn \__driver_draw_color_cmyk_aux:NNnnnnn #1#2#3#4#5#6
19877      {
19878        \use:x
19879          {
19880            \__driver_draw_color_rgb_auxii:nnn
19881              { \fp_eval:n { -100 * ( (#3) * ( 1 - (#6) ) - 1 ) } }
19882              { \fp_eval:n { -100 * ( (#4) * ( 1 - (#6) ) + #6 - 1 ) } }
19883              { \fp_eval:n { -100 * ( (#5) * ( 1 - (#6) ) + #6 - 1 ) } }
19884          }
19885        #1 #2
19886      }
19887    \cs_new_protected:Npn \__driver_draw_color_cmyk:nnnn
19888      { \__driver_draw_color_cmyk_aux:NNnnnnn \c_true_bool \c_true_bool }
19889    \cs_new_protected:Npn \__driver_draw_color_cmyk_fill:nnnn
19890      { \__driver_draw_color_cmyk_aux:NNnnnnn \c_false_bool \c_true_bool }
19891    \cs_new_protected:Npn \__driver_draw_color_cmyk_stroke:nnnn
19892      { \__driver_draw_color_cmyk_aux:NNnnnnn \c_true_bool \c_false_bool }
19893    \cs_new_protected:Npn \__driver_draw_color_gray_aux:NNn #1#2#3
19894      {
19895        \use:x
19896          {
19897            \__driver_draw_color_gray_aux:nNN
19898              { \fp_eval:n { 100 * (#3)} }
19899          }
19900          #1 #2
19901      }
19902    \cs_new_protected:Npn \__driver_draw_color_gray_aux:nNN #1
19903      { \__driver_draw_color_rgb_auxii:nnnNN {#1} {#1} {#1} }
19904    \cs_generate_variant:Nn \__driver_draw_color_gray_aux:nNN { x }
19905    \cs_new_protected:Npn \__driver_draw_color_gray:n
19906      { \__driver_draw_color_gray_aux:NNn \c_true_bool \c_true_bool }
19907    \cs_new_protected:Npn \__driver_draw_color_gray_fill:n
19908      { \__driver_draw_color_gray_aux:NNn \c_false_bool \c_true_bool }
19909    \cs_new_protected:Npn \__driver_draw_color_gray_stroke:n
19910      { \__driver_draw_color_gray_aux:NNn \c_true_bool \c_false_bool }
19911    \cs_new_protected:Npn \__driver_draw_color_rgb_auxi:NNnnn #1#2#3#4#5
19912      {
19913        \use:x
19914          {
19915            \__driver_draw_color_rgb_auxii:nnnNN
19916              { \fp_eval:n { 100 * (#3) } }
19917              { \fp_eval:n { 100 * (#4) } }
```

```
19918          { \fp_eval:n { 100 * (#5) } }
19919        }
19920      #1 #2
19921    }
19922 \cs_new_protected:Npn \__driver_draw_color_rgb_auxii:nnnNN #1#2#3#4#5
19923   {
19924     \__driver_draw_scope:x
19925       {
19926         \bool_if:NT #4
19927           {
19928             fill =
19929               "
19930                 rgb
19931                   (
19932                     #1 \c_percent_str ,
19933                     #2 \c_percent_str ,
19934                     #3 \c_percent_str
19935                   )
19936               "
19937             \bool_if:NT #5 { ~ }
19938           }
19939         \bool_if:NT #5
19940           {
19941             stroke =
19942               "
19943                 rgb
19944                   (
19945                     #1 \c_percent_str ,
19946                     #2 \c_percent_str ,
19947                     #3 \c_percent_str
19948                   )
19949               "
19950           }
19951       }
19952   }
19953 \cs_new_protected:Npn \__driver_draw_color_rgb:nnn
19954   { \__driver_draw_color_rgb_auxi:NNnnn \c_true_bool \c_true_bool }
19955 \cs_new_protected:Npn \__driver_draw_color_rgb_fill:nnn
19956   { \__driver_draw_color_rgb_auxi:NNnnn \c_false_bool \c_true_bool }
19957 \cs_new_protected:Npn \__driver_draw_color_rgb_stroke:nnn
19958   { \__driver_draw_color_rgb_auxi:NNnnn \c_true_bool \c_false_bool }
```

(*End definition for* __driver_draw_color_cmyk:nnnn *and others.*)

__driver_draw_transformcm:nnnnnn   The first four arguments here are floats (the affine matrix), the last two are a displacement
vector. Once again, force evaluation to allow for caching.

```
19959 \cs_new_protected:Npn \__driver_draw_transformcm:nnnnnn #1#2#3#4#5#6
19960   {
19961     \__driver_draw_scope:x
19962       {
19963         transform =
19964           "
19965             matrix
19966               (
```

```
19967                    \fp_eval:n {#1} , \fp_eval:n {#2} ,
19968                    \fp_eval:n {#3} , \fp_eval:n {#4} ,
19969                    \dim_to_decimal:n {#5} , \dim_to_decimal:n {#6}
19970                 )
19971               "
19972          }
19973     }
```

(*End definition for* `\__driver_draw_transformcm:nnnnnn`.)

`\__driver_draw_hbox:Nnnnnnn` No special savings can be made here: simply displace the box inside a scope. As there is nothing to re-box, just make the box passed of zero size.

```
19974 \cs_new_protected:Npn \__driver_draw_hbox:Nnnnnnn #1#2#3#4#5#6#7
19975   {
19976     \__driver_scope_begin:
19977     \__driver_draw_transformcm:nnnnnn {#2} {#3} {#4} {#5} {#6} {#7}
19978     \__driver_literal:n
19979       {
19980         < g~
19981             stroke="none"~
19982             transform="scale(-1,1)~translate({?x},{?y})~scale(-1,-1)"
19983         >
19984       }
19985     \group_begin:
19986       \box_set_wd:Nn #1 { 0pt }
19987       \box_set_ht:Nn #1 { 0pt }
19988       \box_set_dp:Nn #1 { 0pt }
19989       \box_use:N #1
19990     \group_end:
19991     \__driver_literal:n { </g> }
19992     \__driver_scope_end:
19993   }
```

(*End definition for* `\__driver_draw_hbox:Nnnnnnn`.)

```
19994 ⟨/dvisvgm⟩
```

```
19995 ⟨/initex | 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.

C

E

G

Q

quark commands: