The LaTeX3 Sources

The LaTeX3 Project*

May 20, 2014

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 2ε. In time, a LaTeX3 format will be produced based on this code. This allows the code to be used in LaTeX 2ε 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 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.

[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.

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;

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 it 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`.

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 LaTeX 2ε 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 2ε 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 2ε it provides a few functions for setting it up.

`\ExplSyntaxOn`
`\ExplSyntaxOff`

Updated: 2011-08-13

`\ExplSyntaxOn ⟨code⟩ \ExplSyntaxOff`

The `\ExplSyntaxOn` function switches to a category code 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 LaTeX 2ε kernel functions `\ProvidesPackage`, `\ProvidesClass` and `\ProvidesFile`. However, they also implicitly switch `\ExplSyntaxOn` for the remainder of the code with the file. At the end of the file, `\ExplSyntaxOff` will be called to reverse this. (This is the same concept as LaTeX 2ε provides in turning on `\makeatletter` within package and class code.)

`\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 2ε 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

__expl_package_check:

__expl_package_check:

Used to ensure that all parts of expl3 are loaded together (*i.e.* as part of expl3). Issues an error if a kernel package is loaded independently of the bundle.

\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|Npx|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|Npx|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|Npx|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|Npx|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|Npx|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|Npx|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|Npx|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:(cpn|Npx|cpx)`

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

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

`\cs_gset:Npn`
`\cs_gset:(cpn|Npx|cpx)`

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

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

`\cs_gset_nopar:Npn`
`\cs_gset_nopar:(cpn|Npx|cpx)`

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

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

`\cs_gset_protected:Npn`
`\cs_gset_protected:(cpn|Npx|cpx)`

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

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

`\cs_gset_protected_nopar:Npn`
`\cs_gset_protected_nopar:(cpn|Npx|cpx)`

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

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

3.3 Defining new functions using the signature

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

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

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

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

\cs_gset_protected:Nn
\cs_gset_protected:(cn|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 ⟨*cs*$_1$⟩ ⟨*cs*$_2$⟩
\cs_new_eq:NN ⟨*cs*$_1$⟩ ⟨*token*⟩

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

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

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

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

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

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

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

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.

\cs_show:N
\cs_show:c

Updated: 2012-09-09

\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*TF*
\cs_if_exist_use:c*TF*
New: 2012-11-10

`\cs_if_exist_use:N ⟨control sequence⟩`

Tests whether the ⟨*control sequence*⟩ is currently defined (whether as a function or another control sequence type), and if it does inserts the ⟨*control sequence*⟩ into the input stream.

\cs_if_exist_use:N*TF* ⋆
\cs_if_exist_use:c*TF* ⋆
New: 2012-11-10

`\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 does inserts the ⟨*control sequence*⟩ into the input stream followed by the ⟨*true code*⟩.

\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 the situation in force when first function absorbs the token).

`\use:n` ★
`\use:(nn|nnn|nnnn)` ★

`\use:n` {⟨*group*$_1$⟩}
`\use:nn` {⟨*group*$_1$⟩} {⟨*group*$_2$⟩}
`\use:nnn` {⟨*group*$_1$⟩} {⟨*group*$_2$⟩} {⟨*group*$_3$⟩}
`\use:nnnn` {⟨*group*$_1$⟩} {⟨*group*$_2$⟩} {⟨*group*$_3$⟩} {⟨*group*$_4$⟩}

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_ii:nn ★

\use_i:nn {⟨*arg*$_1$⟩} {⟨*arg*$_2$⟩}

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 {⟨*arg*$_1$⟩} {⟨*arg*$_2$⟩} {⟨*arg*$_3$⟩}

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 {⟨*arg*$_1$⟩} {⟨*arg*$_2$⟩} {⟨*arg*$_3$⟩} {⟨*arg*$_4$⟩}

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 {⟨*arg*$_1$⟩} {⟨*arg*$_2$⟩} {⟨*arg*$_3$⟩}

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|nnn|nnnn|nnnnn|nnnnnn|nnnnnnn|nnnnnnnn|nnnnnnnnn) ★

\use_none:n {⟨*group*$_1$⟩}

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	`\use:x {⟨expandable tokens⟩}`
Updated: 2011-12-31	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_stop:w` ⋆
`\use_none_delimit_by_q_recursion_stop:w` ⋆

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

Absorb the ⟨*balanced text*⟩ 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_stop:nw` ⋆
`\use_i_delimit_by_q_recursion_stop:nw` ⋆

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

Absorb the ⟨*balanced text*⟩ 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 2ε. Their use is discouraged in expl3 (although still used in low-level definitions) because they are more fragile and in many cases require more expansion control (hence more code) than the two types of conditionals described above.

`\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:NNTF` ⋆

```
\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:NTF` ⋆
`\cs_if_exist:cTF` ⋆

```
\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:NTF ⋆
\cs_if_free:cTF ⋆

\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 Testing string equality

\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 {⟨tl_1⟩} {⟨tl_2⟩}
\str_if_eq:nnTF {⟨tl_1⟩} {⟨tl_2⟩} {⟨*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 {⟨tl_1⟩} {⟨tl_2⟩}
\str_if_eq_x:nnTF {⟨tl_1⟩} {⟨tl_2⟩} {⟨*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:nnTF ⋆
\str_case:onTF ⋆

New: 2013-07-24

```
\str_case:nnTF {⟨test string⟩}
  {
    {⟨string case_1⟩} {⟨code case_1⟩}
    {⟨string case_2⟩} {⟨code case_2⟩}
    ...
    {⟨string case_n⟩} {⟨code case_n⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

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:nnn {⟨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.

5.3 Engine-specific conditionals

`\luatex_if_engine_p:` ★
`\luatex_if_engine:TF` ★

Updated: 2011-09-06

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

Detects is the document is being compiled using LuaTEX.

`\pdftex_if_engine_p:` ★
`\pdftex_if_engine:TF` ★

Updated: 2011-09-06

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

Detects is the document is being compiled using pdfTEX.

`\xetex_if_engine_p:` ★
`\xetex_if_engine:TF` ★

Updated: 2011-09-06

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

Detects is the document is being compiled using XeTEX.

5.4 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_false: ★
\or: ★
\else: ★
\fi: ★
\reverse_if:N ★

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

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

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 ⟨*arg*$_1$⟩ ⟨*arg*$_2$⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:

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

TEXhackers note: This is TEX's \ifx.

\if:w ★
\if_charcode:w ★
\if_catcode:w ★

\if:w ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:
\if_catcode:w ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ ⟨*true code*⟩ \else: ⟨*false code*⟩ \fi:

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:w ★

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

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

\if_mode_horizontal: ★
\if_mode_vertical: ★
\if_mode_math: ★
\if_mode_inner: ★

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

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

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.

`\__chk_if_exist_var:N`

`\__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.

`\__cs_count_signature:N` ⋆
`\__cs_count_signature:c` ⋆

`\__cs_count_signature:N` ⟨*function*⟩

Splits the ⟨*function*⟩ into the ⟨*name*⟩ (*i.e.* the part before the colon) and the ⟨*signature*⟩ (*i.e.* after the colon). The ⟨*number*⟩ of tokens in the ⟨*signature*⟩ is then left in the input stream. If there was no ⟨*signature*⟩ then the result is the marker value -1.

`\__cs_split_function:NN` ⋆

`\__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).

`\__cs_get_function_name:N` ⋆

`\__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).

`\__cs_get_function_signature:N` ⋆

`\__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).

`\__cs_tmp:w`

Function used for various short-term usages, for instance defining functions whose definition involves tokens which are hard to insert normally (spaces, characters with category other).

`\__kernel_register_show:N`
`\__kernel_register_show:c`

`\__kernel_register_show:N` ⟨*register*⟩

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

`\__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.

`\__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:nn(TF)` coded for speed.

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

Providing variants in this way in style files is uncritical as the `\cs_new_nopar:Npn` function will silently accept definitions whenever the new definition is identical to an already given one. 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`

Updated: 2013-07-09

`\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. If the ⟨*parent control sequence*⟩ is protected then the new 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 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 which 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`, `num`, `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. This recent addition to the argument specifiers may shake things up a bit as most places where `o` is used will be replaced by `V`. The documentation you are currently reading will therefore require a fair bit of re-writing.

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. Let's pretend we want to set the control sequence whose name is given by `b \l_tmpa_tl b` equal to the list of tokens `\aaa a`. Furthermore we want to store the execution of it in a ⟨*tl var*⟩. In this example we assume `\l_tmpa_tl` contains the text string `lur`. The straightforward approach is

```
\tl_set:No \l_tmpb_tl { \tl_set:cn { b \l_tmpa_tl b } { \aaa a } }
```

Unfortunately this only puts `\exp_args:Nc \tl_set:Nn {b \l_tmpa_tl b} { \aaa a }` into `\l_tmpb_tl` and not `\tl_set:Nn \blurb { \aaa a }` as we probably wanted. Using `\tl_set:Nx` is not an option as that will die horribly. Instead we can do a

```
\tl_set:Nf \l_tmpb_tl { \tl_set:cn { b \l_tmpa_tl b } { \aaa a } }
```

which puts the desired result in `\l_tmpb_tl`. It requires `\tl_set:Nf` to be defined as

```
\cs_set_nopar:Npn \tl_set:Nf { \exp_args:NNf \tl_set:Nn }
```

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!

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|NNv|NNV|NNf|Nco|Ncf|Ncc|NVV) ⋆

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

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

\exp_args:Nno ⋆
\exp_args:(NnV|Nnf|Noo|Nof|Noc|Nff|Nfo|Nnc) ⋆

Updated: 2012-01-14

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

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

`\exp_args:NNx`
`\exp_args:(Nnx|Ncx|Nox|Nxo|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|Nccc|NcNc|NcNo|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|Nnno|Nnnc|Nooo)` ⋆

`\exp_args:NNNo ⟨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:(NNox|Nnnx|Nnox|Noox|Ncnx|Nccx)`

`\exp_args:NNnx ⟨token1⟩ ⟨token2⟩ {⟨tokens1⟩} {⟨tokens2⟩}`

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:Nf ⋆
\exp_last_unbraced:(NV|No|Nv|Nco|NcV|NNV|NNo|Nno|Noo|Nfo|NNNV|NNNo|NnNo) ⋆

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

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.

TEXhackers note: As an optimization, the last argument is unbraced by some of those functions before expansion. This can cause problems if the argument is empty: for instance, \exp_last_unbraced:Nf \mypkg_foo: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*⟩ ⟨*tokens*$_1$⟩ {⟨*tokens*$_2$⟩}

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 ⟨*token*$_1$⟩ ⟨*token*$_2$⟩

Carries out a single expansion of ⟨*token*$_2$⟩ (which may consume arguments) prior to the expansion of ⟨*token*$_1$⟩. If ⟨*token*$_2$⟩ is a TEX primitive, it will be executed rather than expanded, while if ⟨*token*$_2$⟩ has not expansion (for example, if it is a character) then it will be left unchanged. It is important to notice that ⟨*token*$_1$⟩ may be *any* single token, including group-opening and -closing tokens ({ or } assuming normal TEX category codes). Unless specifically required, 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 ⋆

\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 ⋆

\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.

\exp_stop_f: ⋆

Updated: 2011-06-03

\function:f ⟨*tokens*⟩ \exp_stop_f: ⟨*more tokens*⟩

This function terminates an f-type expansion. Thus if a function \function: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 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.

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

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

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

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 typical states returned are ⟨*true*⟩ and ⟨*false*⟩ but other states are possible, say an ⟨*error*⟩ state for erroneous input, *e.g.*, text as input in a function comparing integers.

LaTeX3 has two forms of conditional flow processing based on these states. The firs 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. Important to note here is that the arguments are executed after exiting the underlying `\if...\fi:` structure.

1 Defining a set of conditional functions

`\prg_new_conditional:Npnn`
`\prg_new_conditional:Nnn`
`\prg_set_conditional:Npnn`
`\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_new_protected_conditional:Nnn`
`\prg_set_protected_conditional:Npnn`
`\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 will not work properly 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_set_eq_conditional:NNn

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

These functions copies 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_false: ⋆

\prg_return_true:
\prg_return_false:

These 'return' functions define the logical state of a conditional statement. They appear within the code for a conditional function generated by \prg_set_conditional:Npnn, *etc*, to indicate when a true or false branch has been 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).

\bool_new:N
\bool_new:c

\bool_new:N ⟨*boolean*⟩

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

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

\bool_set_false:N ⟨*boolean*⟩

Sets ⟨*boolean*⟩ logically false.

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

\bool_set_true:N ⟨*boolean*⟩

Sets ⟨*boolean*⟩ logically true.

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

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

Sets 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:n(TF), 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

\bool_show:N ⟨*boolean*⟩

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

\bool_show:n

New: 2012-02-09
Updated: 2012-07-08

\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. 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 } ||
    \int_compare_p:n { 1 = \error } % is skipped
  ) &&
! ( \int_compare_p:n { 2 = 4 } )
```

is a valid boolean expression. Note that minimal evaluation is carried out whenever possible so that whenever a truth value cannot be changed any more, the remaining tests within the current group are skipped.

`\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. Minimal evaluation is used in the processing, so that once a result is defined there is not further expansion of the tests. For example

```
\bool_if_p:n
  {
   \int_compare_p:nNn { 1 } = { 1 }
   &&
   (
     \int_compare_p:nNn { 2 } = { 3 } ||
     \int_compare_p:nNn { 4 } = { 4 } ||
     \int_compare_p:nNn { 1 } = { \error } % is skipped
   )
   &&
   ! \int_compare_p:nNn { 2 } = { 4 }
}
```

will be `true` and will not evaluate `\int_compare_p:nNn { 1 } = { \error }`. 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 {⟨*boolexpr*$_1$⟩} {⟨*boolexpr*$_2$⟩}

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:cn ☆

\bool_while_do:Nn ⟨*boolean*⟩ {⟨*code*⟩}

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 ☆

Updated: 2012-07-08

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

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

\bool_do_while:nn ☆

Updated: 2012-07-08

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

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

\bool_until_do:nn ☆

Updated: 2012-07-08

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

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 ☆

Updated: 2012-07-08

\bool_while_do:nn {⟨*boolean expression*⟩} {⟨*code*⟩}

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 n copies

\prg_replicate:nn ★

Updated: 2011-07-04

\prg_replicate:nn {⟨*integer expression*⟩} {⟨*tokens*⟩}

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:*TF* ★

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

Detects if TeX is currently in horizontal mode.

\mode_if_inner_p: ★
\mode_if_inner:*TF* ★

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

Detects if TeX is currently in inner mode.

\mode_if_math_p: ★
\mode_if_math:*TF* ★

Updated: 2011-09-05

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

Detects if TeX is currently in maths mode.

\mode_if_vertical_p: ★
\mode_if_vertical:*TF* ★

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

Detects if TeX is currently in vertical mode.

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_end:` ⋆

Updated: 2011-08-11

```
\group_align_safe_begin:
...
\group_align_safe_end:
```

These functions are used to enclose material in a TeX alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the `&` token inside `\halign`. This is necessary to allow grabbing of tokens for testing purposes, as TeX uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as `\peek_after:Nw` 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.

`\scan_align_safe_stop:`

Updated: 2011-09-06

`\scan_align_safe_stop:`

Stops TeX's scanner looking for expandable control sequences at the beginning of an alignment cell. This function is required, for example, to obtain the expected output when testing `\mode_if_math:TF` at the start of a math array cell: placing `\scan_align_safe_stop:` before `\mode_if_math:TF` will give the correct result. This function does not destroy any kerning if used in other locations, but *does* render functions non-expandable.

TeXhackers note: This is a protected version of `\prg_do_nothing:`, which therefore stops TeX's scanner in the circumstances described without producing any affect on the output.

`\__prg_variable_get_scope:N` ⋆

`\__prg_variable_get_scope:N` ⟨*variable*⟩

Returns the scope (`g` for global, blank otherwise) for the ⟨*variable*⟩.

`\__prg_variable_get_type:N` ⋆

`\__prg_variable_get_type:N` ⟨*variable*⟩

Returns the type of ⟨*variable*⟩ (`tl`, `int`, *etc.*)

`\__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

```
\cs_set:Npn \tmp:w #1#2 \q_stop {#1}
```

`\q_mark`

Used as a marker for delimited arguments when `\q_stop` is already in use.

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 6.

`\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:o`

Updated: 2011-09-06

`\quark_if_recursion_tail_stop:n {⟨token list⟩}`

Tests if the ⟨*token list*⟩ contains only `\q_recursion_tail`, and if so 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:on`

Updated: 2011-09-06

`\quark_if_recursion_tail_stop_do:nn {⟨token list⟩} {⟨insertion⟩}`

Tests if the ⟨*token list*⟩ contains only `\q_recursion_tail`, and if so 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 Clearing quarks away

`\use_none_delimit_by_q_recursion_stop:w`

`\use_none_delimit_by_q_recursion_stop:w ⟨tokens⟩ \q_recursion_stop`

Used to prematurely terminate a recursion using `\q_recursion_stop` as the end marker, removing any remaining ⟨*tokens*⟩ from the input stream.

`\use_i_delimit_by_q_recursion_stop:nw`

`\use_i_delimit_by_q_recursion_stop:nw {⟨insertion⟩} ⟨tokens⟩ \q_recursion_stop`

Used to prematurely terminate a recursion using `\q_recursion_stop` as the end marker, removing any remaining ⟨*tokens*⟩ from the input stream. The ⟨*insertion*⟩ is then made into the input stream after the end of the recursion.

6 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`.

7 Internal quark functions

`\__quark_if_recursion_tail_break:NN`
`\__quark_if_recursion_tail_break:nN`

`\__quark_if_recursion_tail_break:nN {⟨token list⟩}`
`\⟨type⟩_map_break:`

Tests if ⟨*token list*⟩ contains only `\q_recursion_tail`, and if so terminates the recursion using `\⟨type⟩_map_break:`. The recursion end should be marked by `\prg_break_point:Nn \⟨type⟩_map_break:`.

8 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 of the time we will be using the term "token" but most of the time the function we're describing can equally well by used on a control sequence as such one is one token as well.

We shall refer to list of tokens as `tlist`s and such lists represented by a single control sequence is a "token list variable" `tl var`. Functions for these two types are found in the l3tl module.

1 All possible tokens

Let us start by reviewing every case that a given token can fall into. It is very important to distinguish two aspects of a token: its meaning, and what it looks like.

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three for the same internal operation of TeX, namely the primitive testing the next two characters for equality of their character code. They 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
```

2 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*⟩

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*⟩`}`

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`

`\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`

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

This function set up the behaviour of ⟨*character*⟩ when found inside `\tl_to_lowercase:n`, such that ⟨*character*$_1$⟩ will be converted into ⟨*character*$_2$⟩. The two ⟨*characters*⟩ may be specified using an ⟨*integer expression*⟩ for the character code concerned. This may include the TeX ‘⟨*character*⟩ method for converting a single character into its character code:

```
\char_set_lccode:nn { ‘\A } { ‘\a } % Standard behaviour
\char_set_lccode:nn { ‘\A } { ‘\A + 32 }
\char_set_lccode:nn { 50 } { 60 }
```

The setting applies within the current TeX group.

`\char_value_lccode:n` ⋆

`\char_value_lccode:n {⟨integer expression⟩}`

Expands to the current lower case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_lccode:n`

`\char_show_value_lccode:n {⟨integer expression⟩}`

Displays the current lower case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_uccode:nn` `\char_set_uccode:nn {⟨intexpr_1⟩} {⟨intexpr_2⟩}`

This function set up the behaviour of ⟨*character*⟩ when found inside `\tl_to_uppercase:n`, such that ⟨*character*$_1$⟩ will be converted into ⟨*character*$_2$⟩. The two ⟨*characters*⟩ may be specified using an ⟨*integer expression*⟩ for the character code concerned. This may include the TeX ‘⟨*character*⟩ method for converting a single character into its character code:

```
\char_set_uccode:nn { `\a } { `\A } % Standard behaviour
\char_set_uccode:nn { `\A } { `\A - 32 }
\char_set_uccode:nn { 60 } { 50 }
```

The setting applies within the current TeX group.

`\char_value_uccode:n` ⋆ `\char_value_uccode:n {⟨integer expression⟩}`

Expands to the current upper case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_uccode:n` `\char_show_value_uccode:n {⟨integer expression⟩}`

Displays the current upper case code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_mathcode:nn` `\char_set_mathcode:nn {⟨intexpr_1⟩} {⟨intexpr_2⟩}`

This function sets up the math code of ⟨*character*⟩. The ⟨*character*⟩ is specified as an ⟨*integer expression*⟩ which will be used as the character code of the relevant character. The setting applies within the current TeX group.

`\char_value_mathcode:n` ⋆ `\char_value_mathcode:n {⟨integer expression⟩}`

Expands to the current math code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩.

`\char_show_value_mathcode:n` `\char_show_value_mathcode:n {⟨integer expression⟩}`

Displays the current math code of the ⟨*character*⟩ with character code given by the ⟨*integer expression*⟩ on the terminal.

`\char_set_sfcode:nn` `\char_set_sfcode:nn {⟨intexpr_1⟩} {⟨intexpr_2⟩}`

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

Used to track which tokens will require special handling at the document level as they are of category ⟨*active*⟩ (catcode 13). Each entry in the sequence consists of a single active character. Active tokens should be added to the sequence when they are defined for general document use.

`\l_char_special_seq`

New: 2012-01-23

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 ⟨$token_1$⟩ to globally be a snapshot of ⟨$token_2$⟩. This will be an implicit representation of ⟨$token_2$⟩.

`\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:N*TF* ⋆

\token_if_group_begin_p:N ⟨*token*⟩
\token_if_group_begin:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of a begin group token ({ when normal TEX category codes are in force). Note that an explicit begin group token cannot be tested in this way, as it is not a valid N-type argument.

\token_if_group_end_p:N ⋆
\token_if_group_end:N*TF* ⋆

\token_if_group_end_p:N ⟨*token*⟩
\token_if_group_end:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token*⟩ has the category code of an end group token (} when normal TEX category codes are in force). Note that an explicit end group token cannot be tested in this way, as it is not a valid N-type argument.

\token_if_math_toggle_p:N ⋆
\token_if_math_toggle:N*TF* ⋆

\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:N*TF* ⋆

\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:NTF ★

\token_if_letter_p:N ⟨token⟩
\token_if_letter:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if ⟨token⟩ has the category code of a letter token.

\token_if_other_p:N ★
\token_if_other:NTF ★

\token_if_other_p:N ⟨token⟩
\token_if_other:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if ⟨token⟩ has the category code of an “other” token.

\token_if_active_p:N ★
\token_if_active:NTF ★

\token_if_active_p:N ⟨token⟩
\token_if_active:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}

Tests if ⟨token⟩ has the category code of an active character.

\token_if_eq_catcode_p:NN ★
\token_if_eq_catcode:NNTF ★

\token_if_eq_catcode_p:NN ⟨$token_1$⟩ ⟨$token_2$⟩
\token_if_eq_catcode:NNTF ⟨$token_1$⟩ ⟨$token_2$⟩ {⟨true code⟩} {⟨false code⟩}

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

\token_if_eq_charcode_p:NN ★
\token_if_eq_charcode:NNTF ★

\token_if_eq_charcode_p:NN ⟨$token_1$⟩ ⟨$token_2$⟩
\token_if_eq_charcode:NNTF ⟨$token_1$⟩ ⟨$token_2$⟩ {⟨true code⟩} {⟨false code⟩}

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

\token_if_eq_meaning_p:NN ⋆
\token_if_eq_meaning:NNTF ⋆

\token_if_eq_meaning_p:NN ⟨*token*$_1$⟩ ⟨*token*$_2$⟩
\token_if_eq_meaning:NNTF ⟨*token*$_1$⟩ ⟨*token*$_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

\token_if_macro_p:N ⋆
\token_if_macro:NTF ⋆

Updated: 2011-05-23

\token_if_macro_p:N ⟨*token*⟩
\token_if_macro:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is a TeX macro.

\token_if_cs_p:N ⋆
\token_if_cs:NTF ⋆

\token_if_cs_p:N ⟨*token*⟩
\token_if_cs:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is a control sequence.

\token_if_expandable_p:N ⋆
\token_if_expandable:NTF ⋆

\token_if_expandable_p:N ⟨*token*⟩
\token_if_expandable:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is expandable. This test returns ⟨*false*⟩ for an undefined token.

\token_if_long_macro_p:N ⋆
\token_if_long_macro:NTF ⋆

Updated: 2012-01-20

\token_if_long_macro_p:N ⟨*token*⟩
\token_if_long_macro:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

\token_if_protected_macro_p:N ⋆
\token_if_protected_macro:NTF ⋆

Updated: 2012-01-20

\token_if_protected_macro_p:N ⟨*token*⟩
\token_if_protected_macro:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

\token_if_protected_long_macro_p:N ⋆
\token_if_protected_long_macro:NTF ⋆

Updated: 2012-01-20

\token_if_protected_long_macro_p:N ⟨*token*⟩
\token_if_protected_long_macro:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

\token_if_chardef_p:N ⋆
\token_if_chardef:NTF ⋆

Updated: 2012-01-20

\token_if_chardef_p:N ⟨*token*⟩
\token_if_chardef:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token*⟩ is defined to be a chardef.

TeXhackers note: Booleans, boxes and small integer constants are implemented as chardefs.

```
\token_if_mathchardef_p:N ⋆
\token_if_mathchardef:NTF ⋆
```

Updated: 2012-01-20

```
\token_if_mathchardef_p:N ⟨token⟩
\token_if_mathchardef:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*token*⟩ is defined to be a mathchardef.

```
\token_if_dim_register_p:N ⋆
\token_if_dim_register:NTF ⋆
```

Updated: 2012-01-20

```
\token_if_dim_register_p:N ⟨token⟩
\token_if_dim_register:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*token*⟩ is defined to be a dimension register.

```
\token_if_int_register_p:N ⋆
\token_if_int_register:NTF ⋆
```

Updated: 2012-01-20

```
\token_if_int_register_p:N ⟨token⟩
\token_if_int_register:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

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:NTF ⋆
```

New: 2012-02-15

```
\token_if_muskip_register_p:N ⟨token⟩
\token_if_muskip_register:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*token*⟩ is defined to be a muskip register.

```
\token_if_skip_register_p:N ⋆
\token_if_skip_register:NTF ⋆
```

Updated: 2012-01-20

```
\token_if_skip_register_p:N ⟨token⟩
\token_if_skip_register:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*token*⟩ is defined to be a skip register.

```
\token_if_toks_register_p:N ⋆
\token_if_toks_register:NTF ⋆
```

Updated: 2012-01-20

```
\token_if_toks_register_p:N ⟨token⟩
\token_if_toks_register:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

Tests if the ⟨*token*⟩ is defined to be a toks register (not used by LATEX3).

```
\token_if_primitive_p:N ⋆
\token_if_primitive:NTF ⋆
```

Updated: 2011-05-23

```
\token_if_primitive_p:N ⟨token⟩
\token_if_primitive:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}
```

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`
Updated: 2012-12-20

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

Tests if the next ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ 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:NTF`
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:NTF`

Updated: 2012-12-20

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

Tests if the next ⟨*token*⟩ in the input stream has the same category code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ 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:NTF`

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:NTF`

Updated: 2012-12-20

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

Tests if the next ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ 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:NTF`

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:NTF`

Updated: 2012-12-20

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

Tests if the next ⟨*token*⟩ in the input stream has the same character code as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ 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:NTF`
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:NTF`
Updated: 2011-07-02

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

Tests if the next ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ 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:NTF`
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:NTF`
Updated: 2011-07-02

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

Tests if the next ⟨*token*⟩ in the input stream has the same meaning as the ⟨*test token*⟩ (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the ⟨*token*⟩ 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:NTF`
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

`\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

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 `)`. After two expansions, `\int_eval:n` yields an ⟨*integer denotation*⟩ which is left in the input stream. This 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 calculates the result of dividing the first value by the second, rounding any remainder. 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 {⟨*intexpr*$_1$⟩} {⟨*intexpr*$_2$⟩}

Evaluates the two ⟨*integer expressions*⟩ as described earlier, then calculates the result of dividing the first value by the second, truncating any remainder. Note that division using / rounds the result. 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 {⟨*intexpr*$_1$⟩} {⟨*intexpr*$_2$⟩}
\int_min:nn {⟨*intexpr*$_1$⟩} {⟨*intexpr*$_2$⟩}

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 {⟨*intexpr*$_1$⟩} {⟨*intexpr*$_2$⟩}

Evaluates the two ⟨*integer expressions*⟩ as described earlier, then calculates the integer remainder of dividing the first expression by the second. This 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 ⟨*integer*$_1$⟩ ⟨*integer*$_2$⟩

Sets the content of ⟨*integer*$_1$⟩ equal to that of ⟨*integer*$_2$⟩.

\int_if_exist_p:N ⋆
\int_if_exist_p:c ⋆
\int_if_exist: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

\int_compare_p:nNn ⋆
\int_compare:nNnTF ⋆

```
\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	<

\int_compare_p:n ⋆
\int_compare:nTF ⋆

Updated: 2013-01-13

```
\int_compare_p:n
  {
    ⟨intexpr1⟩ ⟨relation1⟩
    ...
    ⟨intexprN⟩ ⟨relationN⟩
    ⟨intexprN+1⟩
  }
\int_compare:nTF
  {
    ⟨intexpr1⟩ ⟨relation1⟩
    ...
    ⟨intexprN⟩ ⟨relationN⟩
    ⟨intexprN+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 ⟨*intexpr*$_1$⟩ and ⟨*intexpr*$_2$⟩ using the ⟨*relation*$_1$⟩, then ⟨*intexpr*$_2$⟩ and ⟨*intexpr*$_3$⟩ using the ⟨*relation*$_2$⟩, 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: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: 2012-06-29

\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*⟩). Thus ⟨*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: 2012-06-29

\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. The ⟨*code*⟩ is then placed in front of each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩). Thus the ⟨*code*⟩ should define a function of one argument (#1).

\int_step_variable:nnnNn

New: 2012-06-04
Updated: 2012-06-29

\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. 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.

`\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`.

\int_to_oct:n ⋆

New: 2014-02-11

`\int_to_octal:n {⟨integer expression⟩}`

Calculates the value of the ⟨*integer expression*⟩ and places the octal (base 8) representation of the result in the input stream.

\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.

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 ⋆

`\int_from_alph:n {⟨letters⟩}`

Converts the ⟨*letters*⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨*letters*⟩ are treated using the English alphabet only, with "a" equal to 1 through to "z" equal to 26. Either lower or upper case letters may be used. This is the inverse function of `\int_to_alph:n`.

\int_from_bin:n ⋆

New: 2014-02-11

`\int_from_bin:n {⟨binary number⟩}`

Converts the ⟨*binary number*⟩ into the integer (base 10) representation and leaves this in the input stream.

\int_from_hex:n ⋆

New: 2014-02-11

`\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.

\int_from_oct:n ⋆

New: 2014-02-11

\int_from_oct:n {⟨*octal number*⟩}

Converts the ⟨*octal number*⟩ into the integer (base 10) representation and leaves this in the input stream.

\int_from_roman:n ⋆

\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*⟩ may be in upper or lower case; if the numeral is not valid then the resulting value will be -1.

\int_from_base:nn ⋆

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

Converts the ⟨*number*⟩ in ⟨*base*⟩ into the appropriate value in base 10. The ⟨*number*⟩ should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum ⟨*base*⟩ value is 36.

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: 2012-05-27

\int_show:n ⟨*integer expression*⟩

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

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.

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 ⟨*integer$_1$*⟩ ⟨*relation*⟩ ⟨*integer$_2$*⟩
  ⟨*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*⟩ ⟨*case$_0$*⟩
  \or: ⟨*case$_1$*⟩
  \or: ...
  \else: ⟨*default*⟩
\fi:

Selects a case to execute based on the value of the ⟨*integer*⟩. The first case (⟨*case$_0$*⟩) is executed if ⟨*integer*⟩ is 0, the second (⟨*case$_1$*⟩) if the ⟨*integer*⟩ is 1, *etc.* The ⟨*integer*⟩ may be a literal, a constant or an integer expression (*e.g.* using \int_eval:n).

TEXhackers note: These are the TEX primitives \ifcase and \or.

\if_int_odd:w ⋆

\if_int_odd:w ⟨*tokens*⟩ ⟨*optional space*⟩
  ⟨*true code*⟩
\else:
  ⟨*true code*⟩
\fi:

Expands ⟨*tokens*⟩ until a non-numeric token or a space is found, and tests whether the resulting ⟨*integer*⟩ is odd. If so, ⟨*true code*⟩ is executed. The \else: branch is optional.

TEXhackers note: This is the TEX primitive \ifodd.

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_end: ⋆

__int_eval:w ⟨*intexpr*⟩ __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:Nw

__prg_compare_error:
__prg_compare_error:Nw ⟨*token*⟩

These are used within \int_compare:n(TF), \dim_compare:n(TF) 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:NTF` ⋆
`\dim_if_exist:cTF` ⋆

New: 2012-03-03

`\dim_if_exist_p:N ⟨dimension⟩`
`\dim_if_exist:NTF ⟨dimension⟩ {⟨true code⟩} {⟨false code⟩}`

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

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:nNnTF ★

```
\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` ⋆

Updated: 2013-01-13

```
\dim_compare_p:n
  {
    ⟨dimexpr1⟩ ⟨relation1⟩
    ...
    ⟨dimexprN⟩ ⟨relationN⟩
    ⟨dimexprN+1⟩
  }
\dim_compare:nTF
  {
    ⟨dimexpr1⟩ ⟨relation1⟩
    ...
    ⟨dimexprN⟩ ⟨relationN⟩
    ⟨dimexprN+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 ⟨$dimexpr_1$⟩ and ⟨$dimexpr_2$⟩ using the ⟨$relation_1$⟩, then ⟨$dimexpr_2$⟩ and ⟨$dimexpr_3$⟩ using the ⟨$relation_2$⟩, 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: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.

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: 2012-05-27

\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 ⟨*skip*$_1$⟩ ⟨*skip*$_2$⟩

Sets the content of ⟨*skip*$_1$⟩ equal to that of ⟨*skip*$_2$⟩.

\skip_sub:Nn
\skip_sub:cn
\skip_gsub:Nn
\skip_gsub:cn

Updated: 2011-10-22

\skip_sub:Nn ⟨*skip*⟩ {⟨*skip 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: 2012-05-27

\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|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|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:N*TF* ⋆
\muskip_if_exist:c*TF* ⋆

New: 2012-03-03

\muskip_if_exist_p:N ⟨*muskip*⟩
\muskip_if_exist:NTF ⟨*muskip*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

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 ⟨*muskip*$_1$⟩ ⟨*muskip*$_2$⟩

Sets the content of ⟨*muskip*$_1$⟩ equal to that of ⟨*muskip*$_2$⟩.

\muskip_sub:Nn
\muskip_sub:cn
\muskip_gsub:Nn
\muskip_gsub:cn

Updated: 2011-10-22

\muskip_sub:Nn ⟨*muskip*⟩ {⟨*muskip 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: 2012-05-27

`\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.

__dim_strip_bp:n ★
__dim_strip_pt:n ★

New: 2011-11-11

__dim_strip_bp:n {⟨*dimension expression*⟩}
__dim_strip_pt: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 magnitude of the result, expressed in big points (bp) or points (pt), will be left in the input stream with *no units*. If the decimal part of the magnitude is zero, this will be omitted.

If the {⟨*dimension expression*⟩} contains additional units, these will be ignored, so for example

```
\__dim_strip_pt:n { 1 bp pt }
```

will leave 1.00374 in the input stream (*i.e.* the magnitude of one "big point" when converted to points).

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.

TEXhackers note: When TEX fetches an undelimited argument from the input stream, explicit character tokens with character code 32 (space) and category code 10 (space), which we here call "explicit space characters", are ignored. If the following token is an explicit character token with category code 1 (begin-group) and an arbitrary character code, then TEX scans ahead to obtain an equal number of explicit character tokens with category code 1 (begin-group) and 2 (end-group), and the resulting list of tokens (with outer braces removed) becomes the argument. Otherwise, a single token is taken as the argument for the macro: we call such single tokens "N-type", as they are suitable to be used as an argument for a function with the signature `:N`.

When TEX reads a character of category code 10 for the first time, it is converted to an explicit space character, with character code 32, regardless of the initial character code. "Funny" spaces with a different category code, can be produced using `\tl_to_lowercase:n` or `\tl_to_uppercase:n`. Explicit space characters are also produced as a result of `\token_to_str:N`, `\tl_to_str:n`, etc.

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* var_1⟩ ⟨*tl* var_2⟩

Sets the content of ⟨*tl* var_1⟩ equal to that of ⟨*tl* var_2⟩.

\tl_concat:NNN
\tl_concat:ccc
\tl_gconcat:NNN
\tl_gconcat:ccc

New: 2012-05-18

\tl_concat:NNN ⟨*tl* var_1⟩ ⟨*tl* var_2⟩ ⟨*tl* var_3⟩

Concatenates the content of ⟨*tl* var_2⟩ and ⟨*tl* var_3⟩ together and saves the result in ⟨*tl* var_1⟩. The ⟨*tl* var_2⟩ 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

\tl_set_rescan:Nnn
\tl_set_rescan:(Nno|Nnx|cnn|cno|cnx)
\tl_gset_rescan:Nnn
\tl_gset_rescan:(Nno|Nnx|cnn|cno|cnx)

Updated: 2011-12-18

\tl_set_rescan:Nnn ⟨*tl var*⟩ {⟨*setup*⟩} {⟨*tokens*⟩}

Sets ⟨*tl var*⟩ to contain ⟨*tokens*⟩, applying the category code régime specified in the ⟨*setup*⟩ before carrying out the assignment. This allows the ⟨*tl var*⟩ to contain material with category codes other than those that apply when ⟨*tokens*⟩ are absorbed. Trailing spaces at the end of the ⟨*tokens*⟩ are discarded in the rescanning process. The ⟨*setup*⟩ is not limited to changes of category code but may contain any valid input, for example assignment of the expansion of active tokens. See also \tl_rescan:nn.

\tl_rescan:nn

Updated: 2011-12-18

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

Rescans ⟨*tokens*⟩ applying the category code régime specified in the ⟨*setup*⟩, and leaves the resulting tokens in the input stream. Trailing spaces at the end of the ⟨*tokens*⟩ are discarded in the rescanning process. The ⟨*setup*⟩ is not limited to changes of category code but may contain any valid input, for example assignment of the expansion of active tokens. See also \tl_set_rescan:Nnn.

5 Reassigning token list character codes

\tl_to_lowercase:n

Updated: 2012-09-08

\tl_to_lowercase:n {⟨*tokens*⟩}

Works through all of the ⟨*tokens*⟩, replacing each character token with the lower case equivalent as defined by \char_set_lccode:nn. Characters with no defined lower case character code are left unchanged. This process does not alter the category code assigned to the ⟨*tokens*⟩.

TEXhackers note: This is a wrapper around the TEX primitive \lowercase.

\tl_to_uppercase:n

Updated: 2012-09-08

\tl_to_uppercase:n {⟨*tokens*⟩}

Works through all of the ⟨*tokens*⟩, replacing each character token with the upper case equivalent as defined by `\char_set_uccode:nn`. Characters with no defined upper case character code are left unchanged. This process does not alter the category code assigned to the ⟨*tokens*⟩.

TeXhackers note: This is a wrapper around the TeX primitive `\uppercase`.

6 Token list conditionals

\tl_if_blank_p:n ⋆
\tl_if_blank_p:(V|o) ⋆
\tl_if_blank:n*TF* ⋆
\tl_if_blank:(V|o)*TF* ⋆

\tl_if_blank_p:n {⟨*token list*⟩}
\tl_if_blank:nTF {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list*⟩ consists only of blank spaces (*i.e.* contains no item). The test is `true` if ⟨*token list*⟩ is zero or more explicit space characters (explicit tokens with character code 32 and category code 10), and is `false` otherwise.

\tl_if_empty_p:N ⋆
\tl_if_empty_p:c ⋆
\tl_if_empty:N*TF* ⋆
\tl_if_empty:c*TF* ⋆

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

Tests if the ⟨*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:n*TF* ⋆
\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:NN*TF* ⋆
\tl_if_eq:(Nc|cN|cc)*TF* ⋆

\tl_if_eq_p:NN {⟨*tl var*$_1$⟩} {⟨*tl var*$_2$⟩}
\tl_if_eq:NNTF {⟨*tl var*$_1$⟩} {⟨*tl var*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Compares the content of 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:nn*TF*

\tl_if_eq:nnTF ⟨*token list*$_1$⟩ {⟨*token list*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token list*$_1$⟩ and ⟨*token list*$_2$⟩ contain the same list of tokens, both in respect of character codes and category codes.

\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 list$_1$⟩} {⟨token list$_2$⟩} {⟨true code⟩} {⟨false code⟩}

Tests if ⟨*token list$_2$*⟩ is found inside ⟨*token list$_1$*⟩. The ⟨*token list$_2$*⟩ cannot contain the tokens {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

\tl_if_single_p:N ⋆
\tl_if_single_p:c ⋆
\tl_if_single:NTF ⋆
\tl_if_single:cTF ⋆

Updated: 2011-08-13

\tl_if_single_p:N ⟨tl var⟩
\tl_if_single:NTF ⟨tl var⟩ {⟨true code⟩} {⟨false code⟩}

Tests if the content of the ⟨*tl var*⟩ consists of a single item, *i.e.* is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to \tl_count:N.

\tl_if_single_p:n ⋆
\tl_if_single:nTF ⋆

Updated: 2011-08-13

\tl_if_single_p:n {⟨token list⟩}
\tl_if_single:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}

Tests if the ⟨*token list*⟩ has exactly one item, *i.e.* is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to \tl_count:n.

\tl_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.

7 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.

8 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;
- 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.

9 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 ⟨$item_1$⟩⟨$item_2$⟩⟨$item_3$⟩ ...⟨$item_n$⟩ becomes ⟨$item_n$⟩...⟨$item_3$⟩⟨$item_2$⟩⟨$item_1$⟩. 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 \exp_not:n, 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 ⟨$item_1$⟩⟨$item_2$⟩⟨$item_3$⟩ ...⟨$item_n$⟩ becomes ⟨$item_n$⟩...⟨$item_3$⟩⟨$item_2$⟩⟨$item_1$⟩. 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 {⟨$item_1$⟩}{⟨$item_2$⟩}{⟨$item_3$⟩} ...{⟨$item_n$⟩} becomes {⟨$item_n$⟩} ... {⟨$item_3$⟩}{⟨$item_2$⟩}{⟨$item_1$⟩}. 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 \exp_not:n, 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 \exp_not:n, 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.

10 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|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|V|v|f) ⋆

Updated: 2012-09-01

\tl_tail:n {⟨*token list*⟩}

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first ⟨*item*⟩ in the ⟨*token list*⟩, and leaves the remaining tokens in the input stream. Thus for example

```
\tl_tail:n { a ~ {bc} d }
```

and

```
\tl_tail:n { ~ a ~ {bc} d }
```

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.

\str_head:n ⋆
\str_tail:n ⋆

New: 2011-08-10

\str_head:n {⟨*token list*⟩}
\str_tail:n {⟨*token list*⟩}

Converts the ⟨*token list*⟩ into a string, as described for \tl_to_str:n. The \str_-head:n function then leaves the first character of this string in the input stream. The \str_tail:n function leaves all characters except the first in the input stream. The first character may be a space. If the ⟨*token list*⟩ argument is entirely empty, nothing is left in the input stream.

\tl_if_head_eq_catcode_p:nN ⋆
\tl_if_head_eq_catcode:nN*TF* ⋆

Updated: 2012-07-09

\tl_if_head_eq_catcode_p:nN {⟨*token list*⟩} ⟨*test token*⟩
\tl_if_head_eq_catcode:nNTF {⟨*token list*⟩} ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ has the same category code as the ⟨*test token*⟩. In the case where the ⟨*token list*⟩ is empty, the test will always be false.

\tl_if_head_eq_charcode_p:nN ⋆
\tl_if_head_eq_charcode_p:fN ⋆
\tl_if_head_eq_charcode:nN*TF* ⋆
\tl_if_head_eq_charcode:fN*TF* ⋆

Updated: 2012-07-09

\tl_if_head_eq_charcode_p:nN {⟨*token list*⟩} ⟨*test token*⟩
\tl_if_head_eq_charcode:nNTF {⟨*token list*⟩} ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ has the same character code as the ⟨*test token*⟩. In the case where the ⟨*token list*⟩ is empty, the test will always be false.

\tl_if_head_eq_meaning_p:nN ⋆
\tl_if_head_eq_meaning:nN*TF* ⋆

Updated: 2012-07-09

\tl_if_head_eq_meaning_p:nN {⟨*token list*⟩} ⟨*test token*⟩
\tl_if_head_eq_meaning:nNTF {⟨*token list*⟩} ⟨*test token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ has the same meaning as the ⟨*test token*⟩. In the case where ⟨*token list*⟩ is empty, the test will always be false.

\tl_if_head_is_group_p:n ⋆
\tl_if_head_is_group:nTF ⋆

New: 2012-07-08

\tl_if_head_is_group_p:n {⟨*token list*⟩}
\tl_if_head_is_group:nTF {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ is an explicit begin-group character (with category code 1 and any character code), in other words, if the ⟨*token list*⟩ starts with a brace group. In particular, the test is false if the ⟨*token list*⟩ starts with an implicit token such as \c_group_begin_token, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

\tl_if_head_is_N_type_p:n ⋆
\tl_if_head_is_N_type:nTF ⋆

New: 2012-07-08

\tl_if_head_is_N_type_p:n {⟨*token list*⟩}
\tl_if_head_is_N_type:nTF {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ is a normal N-type argument. In other words, it is neither an explicit space character (explicit token with character code 32 and category code 10) nor an explicit begin-group character (with category code 1 and any character code). An empty argument yields false, as it does not have a "normal" first token. This function is useful to implement actions on token lists on a token by token basis.

\tl_if_head_is_space_p:n ⋆
\tl_if_head_is_space:nTF ⋆

Updated: 2012-07-08

\tl_if_head_is_space_p:n {⟨*token list*⟩}
\tl_if_head_is_space:nTF {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

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

11 Viewing token lists

\tl_show:N
\tl_show:c

Updated: 2012-09-09

\tl_show:N ⟨*tl var*⟩

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

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

\tl_show:n

Updated: 2012-09-09

\tl_show:n ⟨*token list*⟩

Displays the ⟨*token list*⟩ on the terminal.

TeXhackers note: This is similar to the ε-TeX primitive \showtokens, wrapped to a fixed number of characters per line.

12 Constant token lists

`\c_empty_tl` Constant that is always empty.

`\c_job_name_tl`
Updated: 2011-08-18

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.

`\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 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` ⟨*sequence*$_1$⟩ ⟨*sequence*$_2$⟩

Sets the content of ⟨*sequence*$_1$⟩ equal to that of ⟨*sequence*$_2$⟩.

`\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 ⟨*sequence*$_1$⟩ ⟨*sequence*$_2$⟩ ⟨*sequence*$_3$⟩

Concatenates the content of ⟨*sequence*$_2$⟩ and ⟨*sequence*$_3$⟩ together and saves the result in ⟨*sequence*$_1$⟩. The items in ⟨*sequence*$_2$⟩ will be placed at the left side of the new sequence.

\seq_if_exist_p:N ⋆
\seq_if_exist_p:c ⋆
\seq_if_exist:N*TF* ⋆
\seq_if_exist:c*TF* ⋆

New: 2012-03-03

\seq_if_exist_p:N ⟨*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.

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:NNTF
\seq_pop_left:cNTF

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:NNTF
\seq_gpop_left:cNTF

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:NNTF
\seq_pop_right:cNTF

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:NNTF
\seq_gpop_right:cNTF

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:nn(TF)`.

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:nn(TF).

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:NN*TF*
\seq_get:cN*TF*
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:NN*TF*
\seq_pop:cN*TF*
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:NN*TF*
\seq_gpop:cN*TF*
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 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.

11 Viewing sequences

\seq_show:N
\seq_show:c

Updated: 2012-09-09

\seq_show:N ⟨*sequence*⟩

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

12 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 XIII

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_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 list*$_1$⟩ ⟨*comma list*$_2$⟩

Sets the content of ⟨*comma list*$_1$⟩ equal to that of ⟨*comma list*$_2$⟩.

\clist_concat:NNN
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc

\clist_concat:NNN ⟨*comma list*$_1$⟩ ⟨*comma list*$_2$⟩ ⟨*comma list*$_3$⟩

Concatenates the content of ⟨*comma list*$_2$⟩ and ⟨*comma list*$_3$⟩ together and saves the result in ⟨*comma list*$_1$⟩. The items in ⟨*comma list*$_2$⟩ 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:N*TF* ⋆
\clist_if_exist:c*TF* ⋆

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*⟩ {⟨*item*$_1$⟩,...,⟨*item*$_n$⟩}

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*⟩ {⟨*item*$_1$⟩,...,⟨*item*$_n$⟩}

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

Updated: 2011-09-05

\clist_put_right:Nn ⟨*comma list*⟩ {⟨*item*$_1$⟩,...,⟨*item*$_n$⟩}

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:c
\clist_gremove_duplicates:N
\clist_gremove_duplicates:c

`\clist_remove_duplicates:N ⟨comma list⟩`

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:cn
\clist_gremove_all:Nn
\clist_gremove_all:cn

Updated: 2011-09-06

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

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).

4 Comma list conditionals

\clist_if_empty_p:N ⋆
\clist_if_empty_p:c ⋆
\clist_if_empty:N*TF* ⋆
\clist_if_empty:c*TF* ⋆

`\clist_if_empty_p:N ⟨comma list⟩`
`\clist_if_empty:NTF ⟨comma list⟩ {⟨true code⟩} {⟨false code⟩}`

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

\clist_if_in:NnTF
\clist_if_in:(NV|No|cn|cV|co|nn|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|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|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|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` ★
`\clist_count:(c|n)` ★

New: 2012-07-13

`\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_use:cnnn ⋆

New: 2013-05-26

\clist_use:Nnnn ⟨*clist var*⟩ {⟨*separator between two*⟩}
{⟨*separator between more than two*⟩} {⟨*separator between final two*⟩}

Places the contents of the ⟨*clist var*⟩ in the input stream, with the appropriate ⟨*separator*⟩ between the items. Namely, if the comma list has more than two items, the ⟨*separator between more than two*⟩ is placed between each pair of items except the last, for which the ⟨*separator between final two*⟩ is used. If the comma list has exactly two items, then they are placed in the input stream separated by the ⟨*separator between two*⟩. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error 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_use:cn ⋆

New: 2013-05-26

\clist_use:Nn ⟨*clist var*⟩ {⟨*separator*⟩}

Places the contents of the ⟨*clist var*⟩ in the input stream, with the ⟨*separator*⟩ between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error 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:NNTF`
`\clist_get:cNTF`
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:NNTF`
`\clist_pop:cNTF`
New: 2012-05-14

`\clist_pop:NNTF ⟨sequence⟩ ⟨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:NNTF`
`\clist_gpop:cNTF`
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 Viewing comma lists

\clist_show:N
\clist_show:c

Updated: 2012-09-09

`\clist_show:N` ⟨*comma list*⟩

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

\clist_show:n

Updated: 2012-09-09

`\clist_show:n` {⟨*tokens*⟩}

Displays the entries in the comma list in the terminal.

9 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 XIV
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 list*$_1$⟩ ⟨*property list*$_2$⟩

Sets the content of ⟨*property list*$_1$⟩ equal to that of ⟨*property list*$_2$⟩.

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)

Updated: 2012-07-09

\prop_put:Nnn ⟨*property list*⟩ {⟨*key*⟩} {⟨*value*⟩}

Adds an entry to the ⟨*property list*⟩ which may be accessed using the ⟨*key*⟩ and which has ⟨*value*⟩. 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)

Updated: 2011-08-28

\prop_get:NnN ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the ⟨*property list*⟩, and places this in the ⟨*token list variable*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ then the ⟨*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)

Updated: 2011-08-18

\prop_pop:NnN ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the ⟨*property list*⟩, and places this in the ⟨*token list variable*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ then the ⟨*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)

Updated: 2011-08-18

\prop_gpop:NnN ⟨*property list*⟩ {⟨*key*⟩} ⟨*tl var*⟩

Recovers the ⟨*value*⟩ stored with ⟨*key*⟩ from the ⟨*property list*⟩, and places this in the ⟨*token list variable*⟩. If the ⟨*key*⟩ is not found in the ⟨*property list*⟩ then the ⟨*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.

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:N*TF* ⋆
\prop_if_exist:c*TF* ⋆

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:N*TF* ⋆
\prop_if_empty:c*TF* ⋆

\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:Nn*TF* ⋆
\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: 2012-09-09

\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 XV
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` ⟨box_1⟩ ⟨box_2⟩

Sets the content of ⟨box_1⟩ equal to that of ⟨box_2⟩.

`\box_set_eq_clear:NN`
`\box_set_eq_clear:(cN|Nc|cc)`

`\box_set_eq_clear:NN` ⟨box_1⟩ ⟨box_2⟩

Sets the content of ⟨box_1⟩ within the current TeX group equal to that of ⟨box_2⟩, then clears ⟨box_2⟩ globally.

`\box_gset_eq_clear:NN`
`\box_gset_eq_clear:(cN|Nc|cc)`

`\box_gset_eq_clear:NN` ⟨box_1⟩ ⟨box_2⟩

Sets the content of ⟨box_1⟩ equal to that of ⟨box_2⟩, then clears ⟨box_2⟩. 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:c

\box_wd:N ⟨*box*⟩

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:cn

Updated: 2011-10-22

\box_set_dp:Nn ⟨*box*⟩ {⟨*dimension expression*⟩}

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:cn

Updated: 2011-10-22

\box_set_ht:Nn ⟨*box*⟩ {⟨*dimension expression*⟩}

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:cn

Updated: 2011-10-22

\box_set_wd:Nn ⟨*box*⟩ {⟨*dimension expression*⟩}

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:N*TF* ⋆
\box_if_empty:c*TF* ⋆

\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:N*TF* ⋆
\box_if_horizontal:c*TF* ⋆

\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:N*TF* ⋆
\box_if_vertical:c*TF* ⋆

\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:c
\box_gset_to_last:N
\box_gset_to_last:c

\box_set_to_last:N ⟨*box*⟩

Sets the ⟨*box*⟩ equal to the last item (box) added to the current partial list, removing the item from the list at the same time. When applied to the main vertical list, the ⟨*box*⟩ 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*⟩ ⟨*intexpr*$_1$⟩ ⟨*intexpr*$_2$⟩

Display the contents of ⟨*box*⟩ in the terminal, showing the first ⟨*intexpr*$_1$⟩ items of the box, and descending into ⟨*intexpr*$_2$⟩ group levels.

\box_log:N
\box_log:c

New: 2012-05-11

\box_show: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_show:Nnn ⟨*box*⟩ ⟨*intexpr*$_1$⟩ ⟨*intexpr*$_2$⟩

Writes the contents of ⟨*box*⟩ to the log, showing the first ⟨*intexpr*$_1$⟩ items of the box, and descending into ⟨*intexpr*$_2$⟩ 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.

TEXhackers note: This is the TEX primitive \hbox.

\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:cn
\hbox_gset:Nn
\hbox_gset:cn

\hbox_set:Nn ⟨*box*⟩ {⟨*contents*⟩}

Typesets the ⟨*contents*⟩ at natural width and then stores the result inside the ⟨*box*⟩.

\hbox_set_to_wd:Nnn
\hbox_set_to_wd:cnn
\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cnn

\hbox_set_to_wd:Nnn ⟨*box*⟩ {⟨*dimexpr*⟩} {⟨*contents*⟩}

Typesets the ⟨*contents*⟩ to the width given by the ⟨*dimexpr*⟩ and then stores the result 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:cw
\hbox_set_end:
\hbox_gset:Nw
\hbox_gset:cw
\hbox_gset_end:

\hbox_set:Nw ⟨*box*⟩ ⟨*contents*⟩ \hbox_set_end:

Typesets the ⟨*contents*⟩ at natural width and then stores the result inside the ⟨*box*⟩. In contrast to \hbox_set:Nn this function does not absorb the argument when finding the ⟨*content*⟩, and so can be used in circumstances where the ⟨*content*⟩ may not be a simple argument.

\hbox_unpack:N
\hbox_unpack:c

\hbox_unpack:N ⟨*box*⟩

Unpacks the content of the horizontal ⟨*box*⟩, retaining any stretching or shrinking applied when the ⟨*box*⟩ was set.

TEXhackers note: This is the TEX primitive \unhcopy.

\hbox_unpack_clear:N
\hbox_unpack_clear:c

\hbox_unpack_clear:N ⟨*box*⟩

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 ⟨box_1⟩ ⟨box_2⟩ {⟨dimexpr⟩}`

Sets ⟨box_1⟩ to contain material to the height given by the ⟨*dimexpr*⟩ by removing content from the top of ⟨box_2⟩ (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 XVI
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 ⟨$coffin_1$⟩ ⟨$coffin_2$⟩

Sets both the content and poles of ⟨$coffin_1$⟩ equal to those of ⟨$coffin_2$⟩ 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` ⟨*coffin*$_1$⟩ {⟨*coffin*$_1$*-pole*$_1$⟩} {⟨*coffin*$_1$*-pole*$_2$⟩} ⟨*coffin*$_2$⟩ {⟨*coffin*$_2$*-pole*$_1$⟩} {⟨*coffin*$_2$*-pole*$_2$⟩} {⟨*x-offset*⟩} {⟨*y-offset*⟩}

This function attaches ⟨*coffin*$_2$⟩ to ⟨*coffin*$_1$⟩ such that the bounding box of ⟨*coffin*$_1$⟩ is not altered, *i.e.* ⟨*coffin*$_2$⟩ can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating ⟨*handle*$_1$⟩, the point of intersection of ⟨*coffin*$_1$*-pole*$_1$⟩ and ⟨*coffin*$_1$*-pole*$_2$⟩, and ⟨*handle*$_2$⟩, the point of intersection of ⟨*coffin*$_2$*-pole*$_1$⟩ and ⟨*coffin*$_2$*-pole*$_2$⟩. ⟨*coffin*$_2$⟩ is then attached to ⟨*coffin*$_1$⟩ such that the relationship between ⟨*handle*$_1$⟩ and ⟨*handle*$_2$⟩ is described by the ⟨*x-offset*⟩ and ⟨*y-offset*⟩. The two offsets should be given as dimension expressions.

`\coffin_join:NnnNnnnn`
`\coffin_join:(cnnNnnnn|Nnncnnnn|cnncnnnn)`

`\coffin_join:NnnNnnnn` ⟨*coffin*$_1$⟩ {⟨*coffin*$_1$*-pole*$_1$⟩} {⟨*coffin*$_1$*-pole*$_2$⟩} ⟨*coffin*$_2$⟩ {⟨*coffin*$_2$*-pole*$_1$⟩} {⟨*coffin*$_2$*-pole*$_2$⟩} {⟨*x-offset*⟩} {⟨*y-offset*⟩}

This function joins ⟨*coffin*$_2$⟩ to ⟨*coffin*$_1$⟩ such that the bounding box of ⟨*coffin*$_1$⟩ may expand. The new bounding box will cover the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating ⟨*handle*$_1$⟩, the point of intersection of ⟨*coffin*$_1$*-pole*$_1$⟩ and ⟨*coffin*$_1$*-pole*$_2$⟩, and ⟨*handle*$_2$⟩, the point of intersection of ⟨*coffin*$_2$*-pole*$_1$⟩ and ⟨*coffin*$_2$*-pole*$_2$⟩. ⟨*coffin*$_2$⟩ is then attached to ⟨*coffin*$_1$⟩ such that the relationship between ⟨*handle*$_1$⟩ and ⟨*handle*$_2$⟩ is described by the ⟨*x-offset*⟩ and ⟨*y-offset*⟩. The two offsets should be given as dimension expressions.

`\coffin_typeset:Nnnnn`
`\coffin_typeset:cnnnn`
Updated: 2012-07-20

`\coffin_typeset:Nnnnn` ⟨*coffin*⟩ {⟨*pole*$_1$⟩} {⟨*pole*$_2$⟩} {⟨*x-offset*⟩} {⟨*y-offset*⟩}

Typesetting is carried out by first calculating ⟨*handle*⟩, the point of intersection of ⟨*pole*$_1$⟩ and ⟨*pole*$_2$⟩. The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the ⟨*handle*⟩ is described by the ⟨*x-offset*⟩ and ⟨*y-offset*⟩. The two offsets should be given as dimension expressions. Typesetting a coffin is therefore analogous to carrying out an alignment where the "parent" coffin is the current insertion point.

4 Measuring coffins

`\coffin_dp:N`
`\coffin_dp:c`

`\coffin_dp:N` ⟨*coffin*⟩

Calculates the depth (below the baseline) of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dimension expression*⟩.

\coffin_ht:N
\coffin_ht:c

\coffin_ht:N ⟨*coffin*⟩

Calculates the height (above the baseline) of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dimension expression*⟩.

\coffin_wd:N
\coffin_wd:c

\coffin_wd:N ⟨*coffin*⟩

Calculates the width of the ⟨*coffin*⟩ in a form suitable for use in a ⟨*dimension expression*⟩.

5 Coffin diagnostics

\coffin_display_handles:Nn
\coffin_display_handles:cn
Updated: 2011-09-02

\coffin_display_handles:Nn ⟨*coffin*⟩ {⟨*color*⟩}

This function first calculates the intersections between all of the ⟨*poles*⟩ of the ⟨*coffin*⟩ to give a set of ⟨*handles*⟩. It then prints the ⟨*coffin*⟩ at the current location in the source, with the position of the ⟨*handles*⟩ marked on the coffin. The ⟨*handles*⟩ will be labelled as part of this process: the locations of the ⟨*handles*⟩ and the labels are both printed in the ⟨*color*⟩ specified.

\coffin_mark_handle:Nnnn
\coffin_mark_handle:cnnn
Updated: 2011-09-02

\coffin_mark_handle:Nnnn ⟨*coffin*⟩ {⟨*pole*$_1$⟩} {⟨*pole*$_2$⟩} {⟨*color*⟩}

This function first calculates the ⟨*handle*⟩ for the ⟨*coffin*⟩ as defined by the intersection of ⟨*pole*$_1$⟩ and ⟨*pole*$_2$⟩. It then marks the position of the ⟨*handle*⟩ on the ⟨*coffin*⟩. The ⟨*handle*⟩ 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: 2012-09-09

\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 XVII

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 XVIII
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:nnTF ★

New: 2012-03-03

\msg_if_exist_p:nn {⟨*module*⟩} {⟨*message*⟩}
\msg_if_exist:nnTF {⟨*module*⟩} {⟨*message*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*message*⟩ for the ⟨*module*⟩ is currently defined.

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:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)`
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:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)

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:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)

Updated: 2012-08-11

\msg_error:nnnnnn {⟨*module*⟩} {⟨*message*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Issues ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The error will interrupt processing and issue the text at the terminal. After user input, the run will continue.

\msg_warning:nnnnnn
\msg_warning:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)

Updated: 2012-08-11

\msg_warning:nnxxxx {⟨*module*⟩} {⟨*message*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Issues ⟨*module*⟩ warning ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The warning text will be added to the log file and the terminal, but the TeX run will not be interrupted.

\msg_info:nnnnnn
\msg_info:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)

Updated: 2012-08-11

\msg_info:nnnnnn {⟨*module*⟩} {⟨*message*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Issues ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The information text will be added to the log file.

\msg_log:nnnnnn
\msg_log:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)

Updated: 2012-08-11

\msg_log:nnnnnn {⟨*module*⟩} {⟨*message*⟩} {⟨*arg one*⟩} {⟨*arg two*⟩} {⟨*arg three*⟩} {⟨*arg four*⟩}

Issues ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The information text will be added to the log file: the output is briefer than \msg_info:nnnnnn.

\msg_none:nnnnnn
\msg_none:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)

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

Updated: 2012-04-27

\msg_redirect_class:nn {⟨*class one*⟩} {⟨*class two*⟩}

Changes the behaviour of messages of ⟨*class one*⟩ so that they are processed using the code for those of ⟨*class two*⟩.

\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:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)

Updated: 2012-08-11

__msg_kernel_fatal:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}

Issues kernel ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. After issuing a fatal error the TeX run will halt. Cannot be redirected.

__msg_kernel_error:nnnnnn
__msg_kernel_error:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)

Updated: 2012-08-11

__msg_kernel_error:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}

Issues kernel ⟨*module*⟩ error ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The error will stop processing and issue the text at the terminal. After user input, the run will continue. Cannot be redirected.

__msg_kernel_warning:nnnnnn
__msg_kernel_warning:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)

Updated: 2012-08-11

__msg_kernel_warning:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}

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:(nnnnn|nnnn|nnn|nn|nnxxxx|nnxxx|nnxx|nnx)

Updated: 2012-08-11

__msg_kernel_info:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}

Issues kernel ⟨*module*⟩ information ⟨*message*⟩, passing ⟨*arg one*⟩ to ⟨*arg four*⟩ to the text-creating functions. The information text will be added to the log file.

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|nnnn|nnn|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_term:nnnnnn`
`\__msg_term:(nnnnnV|nnnnn|nnn|nn)`

`\__msg_term:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}`

Prints the ⟨*message*⟩ from ⟨*module*⟩ in the terminal without formatting. Used in messages which print complex variable contents completely.

`\__msg_show_variable:Nnn`

Updated: 2012-09-09

`\__msg_show_variable:Nnn ⟨variable⟩ {⟨type⟩} {⟨formatted content⟩}`

Displays the ⟨*formatted content*⟩ of the ⟨*variable*⟩ of ⟨*type*⟩ in the terminal. The ⟨*formatted content*⟩ will be processed as the first argument in a call to `\iow_wrap:nnnN`, hence `\\`, `\␣` and other formatting sequences can be used. Once expanded and processed, the ⟨*formatted content*⟩ must either be empty or contain `>`; everything until the first `>` will be removed.

`\__msg_show_variable:n`

Updated: 2012-09-09

`\__msg_show_variable:n {⟨formatted text⟩}`

Shows the ⟨*formatted text*⟩ on the terminal. 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:nn`
`\__msg_show_item_unbraced:nn`

Updated: 2012-09-09

`\__msg_show_item:n ⟨item⟩`
`\__msg_show_item:nn ⟨item-key⟩ ⟨item-value⟩`

Auxiliary functions used within the argument of `\__msg_show_variable:Nnn` 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.

Part XIX

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`. 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`

`\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:
  }
```

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.

`.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`
`.choices:on`
`.choices: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 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
.default:o
.default: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 '
  }
```

.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
.initial:o
.initial:x

Updated: 2013-07-09

⟨*key*⟩ .initial:n = {⟨*value*⟩}

Initialises the ⟨*key*⟩ with the ⟨*value*⟩, equivalent to

\keys_set:nn {⟨*module*⟩} { ⟨*key*⟩ = ⟨*value*⟩ }

.int_set:N
.int_set:c
.int_gset:N
.int_gset:c

⟨*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`
`.multichoices:on`
`.multichoices: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.

`.value_forbidden:`

⟨*key*⟩ `.value_forbidden:`

Specifies that ⟨*key*⟩ cannot receive a ⟨*value*⟩ when used. If a ⟨*value*⟩ is given then an error will be issued.

`.value_required:`

⟨*key*⟩ `.value_required:`

Specifies that ⟨*key*⟩ must receive a ⟨*value*⟩ when used. If a ⟨*value*⟩ is not given then an error will be issued.

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.

\l_keys_choice_int
\l_keys_choice_tl

Inside the code block for a choice generated using .choices:nn, the variables \l_keys_-choice_tl and \l_keys_choice_int are available to indicate the name of the current choice, and its position in the comma list. The position is indexed from 1. Note that, as with standard key code generated using .code:n, the value passed to the key (i.e. the choice name) is also available as #1.

On the other hand, it is sometimes useful to create choices which use entirely different code from one another. This can be achieved by setting the .choice: property of a key, then manually defining sub-keys.

```
\keys_define:nn { mymodule }
  {
    key .choice:,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
  }
```

It is possible to mix the two methods, but manually-created choices should *not* use \l_keys_choice_tl or \l_keys_choice_int. These variables do not have defined behaviour when used outside of code created using .choices:nn (*i.e.* anything might happen).

It is possible to allow choice keys to take values which have not previously been defined by adding code for the special unknown choice. The general behavior of the unknown key is described in Section 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 }
      {
        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)`

`\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

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|nn|nV|nv|no)

New: 2011-08-23
Updated: 2014-04-27

\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|nnn|nnV|nnv|nno)`

New: 2013-07-14
Updated: 2014-04-27

`\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

\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 ⋆

\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

\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

\keys_show:nn {⟨module⟩} {⟨key⟩}

Shows the function which is used to actually implement a ⟨*key*⟩ for a ⟨*module*⟩.

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).

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` ⟨*function*$_1$⟩ ⟨*function*$_2$⟩ {⟨*key-value list*⟩}

Parses the ⟨*key–value list*⟩ into a series of ⟨*keys*⟩ and associated ⟨*values*⟩, or keys alone (if no ⟨*value*⟩ was given). ⟨*function*$_1$⟩ should take one argument, while ⟨*function*$_2$⟩ should absorb two arguments. After `\keyval_parse:NNn` has parsed the ⟨*key–value list*⟩, ⟨*function*$_1$⟩ will be used to process keys given with no value and ⟨*function*$_2$⟩ 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 nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the ⟨*key*⟩ and ⟨*value*⟩, then one *outer* set of braces is removed from the ⟨*key*⟩ and ⟨*value*⟩ as part of the processing.

Part XX

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. Spaces are not allowed 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_job_name_tl` at the start of a LATEX run and will be modified each time a file is read using `\file_input:n`.

`\file_if_exist:nTF`

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. Spaces are not allowed in the ⟨*path*⟩.

\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. Spaces are not allowed in the ⟨*path*⟩.

\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:Nn*TF*
\ior_open:cn*TF*

New: 2013-01-12

\ior_open:NnTF ⟨stream⟩ {⟨file name⟩} {⟨true code⟩} {⟨false code⟩}

Opens ⟨*file name*⟩ for reading using ⟨*stream*⟩ as the control sequence for file access. If the ⟨*stream*⟩ was already open it is closed before the new operation begins. The ⟨*stream*⟩ is available for access immediately and will remain allocated to ⟨*file name*⟩ until a `\ior_close:N` instruction is given or the TeX run ends. The ⟨*true code*⟩ is then inserted into the input stream. If the file is not found, no error is raised and the ⟨*false code*⟩ is inserted into the input stream.

\iow_open:Nn
\iow_open: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:c
\iow_close:N
\iow_close:c

Updated: 2012-07-31

`\ior_close:N` ⟨*stream*⟩
`\iow_close:N` ⟨*stream*⟩

Closes the ⟨*stream*⟩. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.

\ior_list_streams:
\iow_list_streams:

Updated: 2012-09-09

`\ior_list_streams:`
`\iow_list_streams:`

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

New: 2012-06-24

`\ior_get:NN` ⟨*stream*⟩ ⟨*token list variable*⟩

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`

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`

`\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`).

\iow_shipout_x:Nn
\iow_shipout_x:Nx

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.

\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).

2.1 Wrapping lines in output

\iow_wrap:nnnN

New: 2012-06-28

\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*⟩. 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 message.

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 context 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).

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

`\l__file_internal_name_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.

`\__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**,

Part XXI

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: $x<y$, $x<=y$, $x>?y$, $x\,!=y$ *etc.*
- Boolean logic: negation $!\,x$, conjunction $x\,\&\&\,y$, disjunction $x\,||\,y$, ternary operator $x\,?\,y:z$.
- 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 $\operatorname{sind} x$, $\operatorname{cosd} x$, $\operatorname{tand} x$, $\operatorname{cotd} x$, $\operatorname{secd} x$, $\operatorname{cscd} x$ expecting their arguments in degrees.
- Inverse trigonometric functions: $\operatorname{asin} x$, $\operatorname{acos} x$, $\operatorname{atan} x$, $\operatorname{acot} x$, $\operatorname{asec} x$, $\operatorname{acsc} x$ giving a result in radians, and $\operatorname{asind} x$, $\operatorname{acosd} x$, $\operatorname{atand} x$, $\operatorname{acotd} x$, $\operatorname{asecd} x$, $\operatorname{acscd} x$ giving a result in degrees.

(not yet) Hyperbolic functions and their inverse functions: $\sinh x$, $\cosh x$, $\tanh x$, $\coth x$, $\operatorname{sech} x$, csch, and $\operatorname{asinh} x$, $\operatorname{acosh} x$, $\operatorname{atanh} x$, $\operatorname{acoth} x$, $\operatorname{asech} x$, $\operatorname{acsch} x$.

- Extrema: $\max(x,y,\ldots)$, $\min(x,y,\ldots)$, $\operatorname{abs}(x)$.
- Rounding functions: $\operatorname{round}(x,n)$ rounds to closest, $\operatorname{trunc}(x,n)$ rounds towards zero, $\operatorname{floor}(x,n)$ rounds towards $-\infty$, $\operatorname{ceil}(x,n)$ rounds towards $+\infty$. 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 var*$_1$⟩ ⟨*fp var*$_2$⟩

Sets the floating point variable ⟨*fp var*$_1$⟩ equal to the current value of ⟨*fp var*$_2$⟩.

\fp_add:Nn
\fp_add:cn
\fp_gadd:Nn
\fp_gadd:cn

Updated: 2012-05-08

\fp_add:Nn ⟨*fp var*⟩ {⟨*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|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|n) ⋆

Updated: 2012-07-08

\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`. 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|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|n) ⋆

New: 2012-05-08
Updated: 2012-07-08

\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:

⟨*optional* `-`⟩⟨*digit*⟩`.`⟨*15 digits*⟩`e`⟨*optional sign*⟩⟨*exponent*⟩

The leading ⟨*digit*⟩ is non-zero except in the case of ± 0. The values $\pm\infty$ and `nan` trigger an "invalid operation" exception.

\fp_to_tl:N ⋆
\fp_to_tl:(c|n) ⋆

Updated: 2012-07-08

\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.

\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 {⟨*fpexpr*$_1$⟩} ⟨*relation*⟩ {⟨*fpexpr*$_2$⟩}
\fp_compare:nNnTF {⟨*fpexpr*$_1$⟩} ⟨*relation*⟩ {⟨*fpexpr*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Compares the ⟨*fpexpr*$_1$⟩ and the ⟨*fpexpr*$_2$⟩, 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
  {
    ⟨fpexpr_1⟩ ⟨relation_1⟩
    ...
    ⟨fpexpr_N⟩ ⟨relation_N⟩
    ⟨fpexpr_N+1⟩
  }
\fp_compare:nTF
  {
    ⟨fpexpr_1⟩ ⟨relation_1⟩
    ...
    ⟨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 ⟨*intexpr*$_1$⟩ and ⟨*intexpr*$_2$⟩ using the ⟨*relation*$_1$⟩, then ⟨*intexpr*$_2$⟩ and ⟨*intexpr*$_3$⟩ using the ⟨*relation*$_2$⟩, 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` ☆ `\fp_while_do:nn { ⟨fpexpr1⟩ ⟨relation⟩ ⟨fpexpr2⟩ } {⟨code⟩}`

New: 2012-08-16

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`.

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, $\mathrm{e} = \exp(1)$.

`\c_pi_fp`

Updated: 2013-11-17

The value of π. This can be input directly in a floating point expression as `pi`.

`\c_one_degree_fp`

New: 2012-05-08
Updated: 2013-11-17

The value of 1° in radians. Multiply an angle given in degrees by this value to obtain a result in radians. Note that trigonometric functions expecting an argument in radians or in degrees are both available. Within floating point expressions, this can be accessed as `deg`.

`\l_tmpa_fp`
`\l_tmpb_fp`

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

`\g_tmpa_fp`
`\g_tmpb_fp`

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

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:nTF` ⋆
New: 2012-08-08

`\fp_if_flag_on_p:n {⟨exception⟩}`
`\fp_if_flag_on:nTF {⟨exception⟩} {⟨true code⟩} {⟨false code⟩}`

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`
New: 2012-08-08

`\fp_flag_off:n {⟨exception⟩}`

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` ⋆
New: 2012-08-08

`\fp_flag_on:n {⟨exception⟩}`

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

New: 2012-07-19
Updated: 2012-08-08

`\fp_trap:nn {⟨exception⟩} {⟨trap type⟩}`

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:(c|n)

New: 2012-05-08
Updated: 2012-08-14

`\fp_show:N ⟨fp var⟩`
`\fp_show:n {⟨floating point expression⟩}`

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:

- ⟨*sign*⟩: a possibly empty string of `+` and `-` characters;
- ⟨*significand*⟩: a non-empty string of digits together with zero or one dot;
- ⟨*exponent*⟩ optionally: the character `e`, followed by a possibly empty string of `+` and `-` tokens, and a non-empty string of digits.

The sign of the resulting number is + if ⟨*sign*⟩ contains an even number of -, and - otherwise, hence, an empty ⟨*sign*⟩ denotes a non-negative input. The stored significand is obtained from ⟨*significand*⟩ by omitting the decimal separator and leading zeros, and rounding to 16 significant digits, filling with trailing zeros if necessary. In particular, the value stored is exact if the input ⟨*significand*⟩ has at most 16 digits. The stored ⟨*exponent*⟩ is obtained by combining the input ⟨*exponent*⟩ (0 if absent) with a shift depending on the position of the significand and the number of leading zeros.

A special case arises if the resulting ⟨*exponent*⟩ is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by $\pm\infty$), or an underflow (resulting in ± 0).

The result is thus ± 0 if and only if ⟨*significand*⟩ 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 ⟨*sign*⟩, 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.

- Implicit multiplication by juxtaposition (`2pi`, `3(4+5)`, *etc*).
- Function calls (`sin`, `ln`, *etc*).
- Binary `**` and `^` (right associative).
- Unary `+`, `-`, `!`.
- Binary `*`, `/` and `%`.
- 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 ⟨*operand*$_2$⟩ if ⟨*operand*$_1$⟩ is true, and ⟨*operand*$_3$⟩ 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.

TWOBARS `\fp_eval:n { ⟨operand1⟩ || ⟨operand2⟩ }`

If ⟨*operand*$_1$⟩ is true (non-zero), use that value, otherwise the value of ⟨*operand*$_2$⟩. Both ⟨*operands*⟩ are evaluated in all cases.

&& `\fp_eval:n { ⟨operand1⟩ && ⟨operand2⟩ }`

If ⟨*operand*$_1$⟩ is false (equal to ± 0), use that value, otherwise the value of ⟨*operand*$_2$⟩. Both ⟨*operands*⟩ are evaluated in all cases.

<
=
>
?

Updated: 2013-12-14

```
\fp_eval:n
  {
    ⟨operand1⟩ ⟨relation1⟩
    ...
    ⟨operandN⟩ ⟨relationN⟩
    ⟨operandN+1⟩
  }
```

Each ⟨*relation*⟩ consists of a non-empty string of `<`, `=`, `>`, and `?`, optionally preceded by `!`, and may not start with `?`. This evaluates to $+1$ if all comparisons ⟨$operand_i$⟩ ⟨$relation_j$⟩ ⟨$operand_{i+1}$⟩ are true, and $+0$ otherwise. All ⟨*operands*⟩ are evaluated in all cases. See `\fp_compare:nTF` for details.

+
-

```
\fp_eval:n { ⟨operand1⟩ + ⟨operand2⟩ }
\fp_eval:n { ⟨operand1⟩ - ⟨operand2⟩ }
```

Computes the sum or the difference of its two ⟨*operands*⟩. The "invalid operation" exception occurs for $\infty - \infty$. "Underflow" and "overflow" occur when appropriate.

*
/

```
\fp_eval:n { ⟨operand1⟩ * ⟨operand2⟩ }
\fp_eval:n { ⟨operand1⟩ / ⟨operand2⟩ }
```

Computes the product or the ratio of its two ⟨*operands*⟩. The "invalid operation" exception occurs for ∞/∞, $0/0$, or $0 * \infty$. "Division by zero" occurs when dividing a finite non-zero number by ± 0. "Underflow" and "overflow" occur when appropriate.

+
-
!

```
\fp_eval:n { + ⟨operand⟩ }
\fp_eval:n { - ⟨operand⟩ }
\fp_eval:n { ! ⟨operand⟩ }
```

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 { ⟨operand1⟩ ** ⟨operand2⟩ }
\fp_eval:n { ⟨operand1⟩ ^ ⟨operand2⟩ }
```

Raises ⟨$operand_1$⟩ to the power ⟨$operand_2$⟩. This operation is right associative, hence `2 ** 2 ** 3` equals $2^{2^3} = 256$. The "invalid operation" exception occurs if ⟨$operand_1$⟩ is negative or -0, and ⟨$operand_2$⟩ 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( ⟨fpexpr⟩ ) }
```

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( ⟨fpexpr⟩ ) }
```

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
min

```
\fp_eval:n { max( ⟨fpexpr1⟩ , ⟨fpexpr2⟩ , ... ) }
\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
trunc
ceil
floor

New: 2013-12-14

```
\fp_eval:n { round ( ⟨fpexpr⟩ ) }
\fp_eval:n { round ( ⟨fpexpr1⟩ , ⟨fpexpr2⟩ ) }
```

Evaluates ⟨*fpexpr*$_1$⟩ $= x$ and ⟨*fpexpr*$_2$⟩ $= n$, 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 ⟨*fpexpr*$_2$⟩ is omitted, $n = 0$, *i.e.*, ⟨*fpexpr*$_1$⟩ is rounded to an integer. The rounding direction depends on the function:

- `round` yields the multiple of 10^{-n} closest to x, and if x is half-way between two such multiples, the even multiple is chosen ("ties to even");
- `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
cos
tan
cot
csc
sec

Updated: 2013-11-17

```
\fp_eval:n { sin( ⟨fpexpr⟩ ) }
\fp_eval:n { cos( ⟨fpexpr⟩ ) }
\fp_eval:n { tan( ⟨fpexpr⟩ ) }
\fp_eval:n { cot( ⟨fpexpr⟩ ) }
\fp_eval:n { csc( ⟨fpexpr⟩ ) }
\fp_eval:n { sec( ⟨fpexpr⟩ ) }
```

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 $\mathrm{sind}(8 \times 180)$ 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` `cosd` `tand` `cotd` `cscd` `secd`	`\fp_eval:n { sind( ⟨fpexpr⟩ ) }`
	`\fp_eval:n { cosd( ⟨fpexpr⟩ ) }`
	`\fp_eval:n { tand( ⟨fpexpr⟩ ) }`
	`\fp_eval:n { cotd( ⟨fpexpr⟩ ) }`
	`\fp_eval:n { cscd( ⟨fpexpr⟩ ) }`
	`\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 $\text{sind}(8 \times 180)$ 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` `acos` `acsc` `asec`	`\fp_eval:n { asin( ⟨fpexpr⟩ ) }`
	`\fp_eval:n { acos( ⟨fpexpr⟩ ) }`
	`\fp_eval:n { acsc( ⟨fpexpr⟩ ) }`
	`\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` `acosd` `acscd` `asecd`	`\fp_eval:n { asind( ⟨fpexpr⟩ ) }`
	`\fp_eval:n { acosd( ⟨fpexpr⟩ ) }`
	`\fp_eval:n { acscd( ⟨fpexpr⟩ ) }`
	`\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 (⟨*fpexpr*$_2$⟩, ⟨*fpexpr*$_1$⟩): this is the arctangent of ⟨*fpexpr*$_1$⟩/⟨*fpexpr*$_2$⟩, possibly shifted by π depending on the signs of ⟨*fpexpr*$_1$⟩ and ⟨*fpexpr*$_2$⟩. The two-argument arccotangent computes the angle in polar coordinates of the point (⟨*fpexpr*$_1$⟩, ⟨*fpexpr*$_2$⟩), equal to the arccotangent of ⟨*fpexpr*$_1$⟩/⟨*fpexpr*$_2$⟩, possibly shifted by π. Both two-argument functions take values in the wider range $[-\pi, \pi]$. The ratio ⟨*fpexpr*$_1$⟩/⟨*fpexpr*$_2$⟩ need not be defined for the two-argument arctangent: when both expressions yield ± 0, or when both yield $\pm\infty$, the resulting angle is one of $\{\pm\pi/4, \pm 3\pi/4\}$ depending on signs. 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 (⟨*fpexpr*$_2$⟩, ⟨*fpexpr*$_1$⟩): this is the arctangent of ⟨*fpexpr*$_1$⟩/⟨*fpexpr*$_2$⟩, possibly shifted by 180 depending on the signs of ⟨*fpexpr*$_1$⟩ and ⟨*fpexpr*$_2$⟩. The two-argument arccotangent computes the angle in polar coordinates of the point (⟨*fpexpr*$_1$⟩, ⟨*fpexpr*$_2$⟩), equal to the arccotangent of ⟨*fpexpr*$_1$⟩/⟨*fpexpr*$_2$⟩, possibly shifted by 180. Both two-argument functions take values in the wider range $[-180, 180]$. The ratio ⟨*fpexpr*$_1$⟩/⟨*fpexpr*$_2$⟩ need not be defined for the two-argument arctangent: when both expressions yield ± 0, or when both yield $\pm\infty$, the resulting angle is one of $\{\pm 45, \pm 135\}$ depending on signs. 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 $\sqrt{\texttt{nan}} = \texttt{nan}$.

inf
nan

The special values $+\infty$, $-\infty$, and `nan` are represented as `inf`, `-inf` and `nan` (see `\c_inf_fp`, `\c_minus_inf_fp` and `\c_nan_fp`).

pi

The value of π (see `\c_pi_fp`).

deg

The value of 1° in radians (see `\c_one_degree_fp`).

em
ex
in
pt
pc
cm
mm
dd
cc
nd
nc
bp
sp

Those units of measurement are equal to their values in `pt`, namely

$$\begin{aligned}
1\texttt{in} &= 72.27\texttt{pt}\\
1\texttt{pt} &= 1\texttt{pt}\\
1\texttt{pc} &= 12\texttt{pt}\\
1\texttt{cm} &= \frac{1}{2.54}\texttt{in} = 28.45275590551181\texttt{pt}\\
1\texttt{mm} &= \frac{1}{25.4}\texttt{in} = 2.845275590551181\texttt{pt}\\
1\texttt{dd} &= 0.376065\texttt{mm} = 1.07000856496063\texttt{pt}\\
1\texttt{cc} &= 12\texttt{dd} = 12.84010277952756\texttt{pt}\\
1\texttt{nd} &= 0.375\texttt{mm} = 1.066978346456693\texttt{pt}\\
1\texttt{nc} &= 12\texttt{nd} = 12.80374015748031\texttt{pt}\\
1\texttt{bp} &= \frac{1}{72}\texttt{in} = 1.00375\texttt{pt}\\
1\texttt{sp} &= 2^{-16}\texttt{pt} = 1.52587890625e-5\texttt{pt}.
\end{aligned}$$

The values of the (font-dependent) units `em` and `ex` are gathered from TeX when the surrounding floating point expression is evaluated.

true
false

Other names for 1 and +0.

`\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.

`\fp_abs:n` ⋆

New: 2012-05-14
Updated: 2012-07-08

`\fp_abs:n {⟨floating point expression⟩}`

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_min:nn` ⋆

New: 2012-09-26

`\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 XXII
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 `\luatex_if_engine:T` to avoid this. Details of coding the LuaTEX engine are detailed in the LuaTEX manual.

`\lua_now:n` ⋆
`\lua_now:x` ⋆
Updated: 2012-08-02

`\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.

`\lua_now_x:n` ⋆
`\lua_now_x:x` ⋆
New: 2012-08-02

`\lua_now_x:n {⟨token list⟩}`

The ⟨*token list*⟩ is first tokenized and expanded 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_x:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the ⟨*Lua input*⟩ immediately, and in an expandable manner.

TEXhackers note: `\lua_now_x:n` is the LuaTEX primitive `\directlua` renamed.

`\lua_shipout:n`
`\lua_shipout:x`

`\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.

TEXhackers note: At a TEX level, the ⟨*Lua input*⟩ is stored as a "whatsit".

\lua_shipout_x:n
\lua_shipout_x:x

`\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: the ⟨*Lua input*⟩ is expanded during this process in addition to any expansion when the argument was read. This makes these functions suitable for including material finalised during the page building process (such as the page number).

TEXhackers note: `\lua_shipout_x:n` is the LuaTEX primitive `\latelua` named using the LATEX3 scheme.

At a TEX level, the ⟨*Lua input*⟩ is stored as a "whatsit".

2 Category code tables

As well as providing methods to break out into Lua, there are places where additional LATEX3 functions are provided by the LuaTEX engine. In particular, LuaTEX provides category code tables. These can be used to ensure that a set of category codes are in force in a more robust way than is possible with other engines. These are therefore used by `\ExplSyntaxOn` and ExplSyntaxOff when using the LuaTEX engine.

\cctab_new:N

`\cctab_new:N ⟨category code table⟩`

Creates a new category code table, initially with the codes as used by iniTEX.

\cctab_gset:Nn

`\cctab_gset:Nn ⟨category code table⟩ {⟨category code set up⟩}`

Sets the ⟨*category code table*⟩ to apply the category codes which apply when the prevailing régime is modified by the ⟨*category code set up*⟩. Thus within a standard code block the starting point will be the code applied by `\c_code_cctab`. The assignment of the table is global: the underlying primitive does not respect grouping.

\cctab_begin:N

`\cctab_begin:N ⟨category code table⟩`

Switches the category codes in force to those stored in the ⟨*category code table*⟩. The prevailing codes before the function is called are added to a stack, for use with `\cctab_end:`.

\cctab_end:

`\cctab_end:`

Ends the scope of a ⟨*category code table*⟩ started using `\cctab_begin:N`, retuning the codes to those in force before the matching `\cctab_begin:N` was used.

\c_code_cctab

Category code table for the code environment. This does not include setting the behaviour of the line-end character, which is only altered by `\ExplSyntaxOn`.

\c_document_cctab Category code table for a standard LaTeX document. This does not include setting the behaviour of the line-end character, which is only altered by \ExplSyntaxOff.

\c_initex_cctab Category code table as set up by iniTeX.

\c_other_cctab Category code table where all characters have category code 12 (other).

\c_str_cctab Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space).

Part XXIII
The l3candidates package
Experimental additions to l3kernel

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.

1 Additions to l3basics

\cs_if_exist_use:NTF ⋆
\cs_if_exist_use:cTF ⋆

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

If the ⟨*control sequence*⟩ exists, leave it in the input stream, followed by the ⟨*true code*⟩ (unbraced). Otherwise, leave the ⟨*false*⟩ code in the input stream. For example,

```
\cs_set:Npn \mypkg_use_character:N #1
  { \cs_if_exist_use:cF { mypkg_#1:n } { \mypkg_default:N #1 } }
```

calls the function `\mypkg_#1:n` if it exists, and falls back to a default action otherwise. This could also be done (more slowly) using `\str_case_x:nnn`.

TEXhackers note: The `c` variants do not introduce the ⟨*control sequence*⟩ in the hash table if it is not there.

2 Additions to l3box

2.1 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. 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. 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. The resizing applies within the current TEX group level.

\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. The scaling applies within the current TEX group level.

2.2 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.

2.3 Internal variables

`\l__box_angle_fp`

The angle through which a box is rotated by `\box_rotate:Nn`, given in degrees counter-clockwise. This value is required by the underlying driver code in l3driver to carry out the driver-dependent part of box rotation.

`\l__box_cos_fp`
`\l__box_sin_fp`

The sine and cosine of the angle through which a box is rotated by `\box_rotate:Nn`: the values refer to the angle counter-clockwise. These values are required by the underlying driver code in l3driver to carry out the driver-dependent part of box rotation.

`\l__box_scale_x_fp`
`\l__box_scale_y_fp`

The scaling factors by which a box is scaled by `\box_scale:Nnn` or `\box_resize:Nnn`. These values are required by the underlying driver code in l3driver to carry out the driver-dependent part of box rotation.

`\l__box_internal_box`

Box used for affine transformations, which is used to contain rotated material when applying `\box_rotate:Nn`. This box must be correctly constructed for the driver-dependent code in l3driver to function correctly.

3 Additions to l3clist

`\clist_item:Nn` ★
`\clist_item:(cn|nn)` ★

`\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.

`\clist_set_from_seq:NN`
`\clist_set_from_seq:(cN|Nc|cc)`
`\clist_gset_from_seq:NN`
`\clist_gset_from_seq:(cN|Nc|cc)`

`\clist_set_from_seq:NN ⟨comma list⟩ ⟨sequence⟩`

Sets the ⟨*comma list*⟩ to be equal to the content of the ⟨*sequence*⟩. Items which contain either spaces or commas are surrounded by braces.

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

`\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_if_empty_p:n` ★
`\clist_if_empty:nTF` ★

`\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.

4 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.

5 Additions to l3file

\ior_map_inline:Nn

New: 2012-02-11

\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

New: 2012-02-11

\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:

New: 2012-06-29

\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

New: 2012-06-29

\ior_map_break:n {⟨*tokens*⟩}

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.

6 Additions to l3fp

\fp_set_from_dim:Nn
\fp_set_from_dim:cn
\fp_gset_from_dim:Nn
\fp_gset_from_dim:cn

\fp_set_from_dim:Nn ⟨*floating point variable*⟩ {⟨*dimexpr*⟩}

Sets the ⟨*floating point variable*⟩ to the distance represented by the ⟨*dimension expression*⟩ in the units points. This means that distances given in other units are first converted to points before being assigned to the ⟨*floating point variable*⟩.

`\fp_function:Nw` ⋆

```
\fp_eval:n
  {
    \fp_function:Nw ⟨function⟩
      ( ⟨fpexpr1⟩ , ... , ⟨fpexprn⟩ )
  }
```

Finds one or more arguments ⟨*fpexpr*$_1$⟩ ... ⟨*fpexpr*$_n$⟩ following the ⟨*function*⟩, and evaluate them. Then calls the ⟨*function*⟩ followed by a single brace group containing {⟨*result*$_1$⟩} ... {⟨*result*$_n$⟩}. For instance,

```
\cs_new_nopar:Npn \mypkg_log:w
  { \fp_function:Nw \__mypkg_log:n }
\cs_new:Npn \__mypkg_log:n #1
  {
    \int_case:nnF { \tl_count:n {#1} }
      {
        { 1 } { \__mypkg_log_aux:nn #1 { 10 } }
        { 2 } { \__mypkg_log_aux:nn #1 }
      }
      { \ERROR \c_nan_fp }
  }
\cs_new:Npn \__mypkg_log_aux:nn #1#2 { ln(#1) / ln(#2) }
\fp_show:n { \mypkg_log:w (8, 2) + \mypkg_log:w (1e17) }
```

shows $20 = \log_2(8) + \log(10^{17})$. The function `\mypkg_log:w` behaves like other built-in functions such as `ln`, but allows 1 or 2 arguments, and computes either the base 10 logarithm or the logarithm of the first argument in a base given by the second argument. Checking the number of arguments is acheived by `\__mypkg_log:n`, which provides the default base 10 when there is only one argument. The computation itself is done by `\__mypkg_log_aux:nn`.

`\fp_new_function:Npn`

`\fp_new_function:Npn` ⟨*function*⟩ ⟨*parameters*⟩ {⟨*code*⟩}
`\fp_eval:n` { ⟨*function*⟩ (⟨*fpexpr*$_1$⟩ , ... , ⟨*fpexpr*$_n$⟩) }

Defines the ⟨*function*⟩ for use within floating point expressions, expecting some ⟨*parameters*⟩, and evaluating the ⟨*code*⟩, which must be expandable. When the ⟨*function*⟩ appears in a floating point expression, arguments ⟨*fpexpr*$_1$⟩, ..., ⟨*fpexpr*$_n$⟩ are found and evaluated in the same way as for built-in functions such as `max`. If the number of arguments matches the number of ⟨*parameters*⟩, the arguments replace `#1`, ..., `#`n in the ⟨*code*⟩, which is then evaluated to produce a floating point result. Otherwise, the result is `nan` after an error. The ⟨*parameter text*⟩ must not contain delimited arguments, that is, it must be empty or one of `#1`, `#1#2`, `#1#2#3`, ... `#1#2#3#4#5#6#7#8#9`. The arguments replacing parameters in the ⟨*code*⟩ are internal floating point numbers; operations such as `#1^2` thus correctly take into account the sign of `#1`. For instance,

```
\fp_new_function:Npn \mypkg_sqrt:w #1 { #1^.5 }
\fp_new_function:Npn \mypkg_veclen:w #1#2
  { \mypkg_sqrt:w ( #1^2 + #2^2 ) }
\fp_show:n { \mypkg_veclen:w ( 42 / 7 , 2 * 4 - 0 ) }
```

shows 10. In the example, `\mypkg_veclen:w` receives the arguments $6 = 42/7$ and $8 = 2 \times 4 - 0$, thus expands to `\mypkg_sqrt:w ( 6^2 + 8^2 )`, then `\mypkg_sqrt:w` receives $100 = 6^2 + 8^2$ as an argument, and evaluates the square root $10 = 100^{0.5}$.

7 Additions to l3prop

`\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_get:Nn` is faster.

`\prop_get:Nn` ★
`\prop_get:cn` ★

`\prop_get: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.

8 Additions to l3seq

\seq_item:Nn ⋆
\seq_item:cn ⋆

\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.

\seq_mapthread_function:NNN ☆
\seq_mapthread_function:(NcN|cNN|ccN) ☆

\seq_mapthread_function:NNN ⟨seq₁⟩ ⟨seq₂⟩ ⟨function⟩

Applies ⟨*function*⟩ to every pair of items ⟨seq_1*-item*⟩–⟨seq_2*-item*⟩ from the two sequences, returning items from both sequences from left to right. The ⟨*function*⟩ 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_from_clist:NN
\seq_set_from_clist:(cN|Nc|cc|Nn|cn)
\seq_gset_from_clist:NN
\seq_gset_from_clist:(cN|Nc|cc|Nn|cn)

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

Sets the ⟨*sequence*⟩ within the current TeX group to be equal to the content of the ⟨*comma-list*⟩.

\seq_reverse:N
\seq_greverse:N

\seq_reverse:N ⟨sequence⟩

Reverses the order of items in the ⟨*sequence*⟩, and assigns the result to ⟨*sequence*⟩, locally or globally according to the variant chosen.

\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 ⟨$sequence_2$⟩. 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 ⟨$sequence_1$⟩.

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` ⟨*sequence*$_1$⟩ ⟨*sequence*$_2$⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*sequence*$_2$⟩. 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 ⟨*sequence*$_1$⟩. As such, the code in ⟨*inline function*⟩ should be expandable.

TEXhackers note: Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and will lead to low-level TEX errors.

9 Additions to l3skip

`\dim_to_pt:n` ⋆

New: 2013-05-06

`\dim_to_pt: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.

If the ⟨*dimension expression*⟩ contains additional tokens such as redundant units, these will be ignored, so for example

```
\dim_to_pt:n { 1 bp pt }
```

leaves `1.00374` in the input stream, *i.e.* the magnitude of one "big point" when converted to points.

`\dim_to_unit:nn` ⋆

New: 2013-05-06

`\dim_to_unit:nn` {⟨*dimexpr*$_1$⟩} {⟨*dimexpr*$_2$⟩}

Evaluates the ⟨*dimension expressions*⟩, and leaves the value of ⟨*dimexpr*$_1$⟩, expressed in a unit given by ⟨*dimexpr*$_2$⟩, 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.

If the ⟨*dimension expressions*⟩ contain additional tokens such as redundant units, these will be ignored, so for example

```
\dim_to_unit:nn { 1 bp pt } { 1 mm }
```

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

`\skip_split_finite_else_action:nnNN`

`\skip_split_finite_else_action:nnNN` {⟨*skipexpr*⟩} {⟨*action*⟩} ⟨*dimen*$_1$⟩ ⟨*dimen*$_2$⟩

Checks if the ⟨*skipexpr*⟩ contains finite glue. If it does then it assigns ⟨*dimen*$_1$⟩ the stretch component and ⟨*dimen*$_2$⟩ the shrink component. If it contains infinite glue set ⟨*dimen*$_1$⟩ and ⟨*dimen*$_2$⟩ to 0 pt and place `#2` into the input stream: this is usually an error or warning message of some sort.

10 Additions to l3tl

\tl_if_single_token_p:n ⋆
\tl_if_single_token:nTF ⋆

`\tl_if_single_token_p:n {⟨token list⟩}`
`\tl_if_single_token:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}`

Tests if the token list consists of exactly one token, *i.e.* is either a single space character or a single "normal" token. Token groups (`{...}`) are not single tokens.

\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_expandable_uppercase:n ⋆
\tl_expandable_lowercase:n ⋆

`\tl_expandable_uppercase:n {⟨tokens⟩}`
`\tl_expandable_lowercase:n {⟨tokens⟩}`

The `\tl_expandable_uppercase:n` function works through all of the ⟨*tokens*⟩, replacing characters in the range `a`–`z` (with arbitrary category code) by the corresponding letter in the range `A`–`Z`, with category code 11 (letter). Similarly, `\tl_expandable_lowercase:n` replaces characters in the range `A`–`Z` by letters in the range `a`–`z`, and leaves other tokens unchanged. This function requires two steps of expansion.

TEXhackers note: Begin-group and end-group characters are normalized and become `{` and `}`, respectively. 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_item:nn ★
\tl_item:(Nn|cn) ★

\tl_item:nn {⟨*token list*⟩} {⟨*integer expression*⟩}

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 Additions to l3tokens

\char_set_active:Npn
\char_set_active:Npx

\char_set_active:Npn ⟨*char*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

Makes ⟨*char*⟩ an active character to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed. The ⟨*char*⟩ is made active within the current TeX group level, and the definition is also local.

\char_gset_active:Npn
\char_gset_active:Npx

\char_gset_active:Npn ⟨*char*⟩ ⟨*parameters*⟩ {⟨*code*⟩}

Makes ⟨*char*⟩ an active character to expand to ⟨*code*⟩ as replacement text. Within the ⟨*code*⟩, the ⟨*parameters*⟩ (#1, #2, *etc.*) will be replaced by those absorbed. The ⟨*char*⟩ is made active within the current TeX group level, but the definition is global. This function is therefore suited to cases where an active character definition should be applied only in some context (where the ⟨*char*⟩ is again made active).

\char_set_active_eq:NN

\char_set_active_eq:NN ⟨*char*⟩ ⟨*function*⟩

Makes ⟨*char*⟩ an active character equivalent in meaning to the ⟨*function*⟩ (which may itself be an active character). The ⟨*char*⟩ is made active within the current TeX group level, and the definition is also local.

\char_gset_active_eq:NN

\char_gset_active_eq:NN ⟨*char*⟩ ⟨*function*⟩

Makes ⟨*char*⟩ an active character equivalent in meaning to the ⟨*function*⟩ (which may itself be an active character). The ⟨*char*⟩ is made active within the current TeX group level, but the definition is global. This function is therefore suited to cases where an active character definition should be applied only in some context (where the ⟨*char*⟩ is again made active).

\peek_N_type:*TF*

Updated: 2012-12-20

`\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 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.
- `xdvipdfmx`: The driver used by XeTeX.

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": several variable values must be in the correct locations for the driver code to function.

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_rotate_begin:`
`\__driver_box_rotate_end:`

New: 2011-09-01
Updated: 2013-12-27

```
\__driver_box_rotate_begin:
\box_use:N \l__box_internal_box
\__driver_box_rotate_end:
```

Rotates the ⟨*box material*⟩ anti-clockwise around the current insertion point. The angle of rotation (in degrees counter-clockwise) and the sine and cosine of this angle should be stored in `\l__box_angle_fp`, `\l__box_sin_fp` and `\l__box_cos_fp`, respectively. Typically, the box material inserted between the beginning and end markers will be stored in `\l__box_internal_box`: this fact is required by some drivers to obtain the correct output.

`\__driver_box_scale_begin:`
`\__driver_box_scale_end:`

New: 2011-09-02
Updated: 2013-12-27

```
\__driver_box_scale_begin:
⟨box material⟩
\__driver_box_scale_end:
```

Scales the ⟨*box material*⟩ (which should be either a `\box_use:N` or `\hbox:n` construct). The ⟨*box material*⟩ is scaled by the values stored in `\l__box_scale_x_fp` and `\l__box_scale_y_fp` in the horizontal and vertical directions, respectively. This function is also reused when resizing boxes: at a driver level, only scalings are supported and so the higher-level code must convert the absolute sizes to scale factors.

3 Color support

`\__driver_color_ensure_current:`

New: 2011-09-03
Updated: 2012-05-18

```
\__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.

Part XXV
Implementation

1 l3bootstrap implementation

```
⟨*initex | package⟩
⟨@@=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.

```
⟨*initex⟩
\catcode `\{ = 1 \relax
\catcode `\} = 2 \relax
\catcode `\# = 6 \relax
\catcode `\^ = 7 \relax
⟨/initex⟩
```

Tab characters should not show up in the code, but to be on the safe side.

```
⟨*initex⟩
\catcode `\^^I = 10 \relax
⟨/initex⟩
```

For LuaTeX, the extra primitives need to be enabled. This is not needed in package mode: plain TeX and ConTeXt have the primitives enabled while LaTeX 2ε has them with the prefix `luatex` (which is handled in l3names).

```
⟨*initex⟩
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname directlua\endcsname\relax
  \directlua{tex.enableprimitives ('', tex.extraprimitives ( ))}
\fi
⟨/initex⟩
```

1.2 The \pdfstrcmp primitive with XeTeX and LuaTeX

Only pdfTeX has a primitive called `\pdfstrcmp`. The XeTeX version is just `\strcmp`, so there is some shuffling to do. As this is still a real primitive, using the pdfTeX name is "safe".

```
\begingroup\expandafter\expandafter\expandafter\endgroup
  \expandafter\ifx\csname pdfstrcmp\endcsname\relax
  \let\pdfstrcmp\strcmp
\fi
```

`\expl_pdfstrcmp` If LuaTeX is in use then no primitive `\(pdf)strcmp` is available. However, it can be emulated using some Lua code. Since this is not the primitive, we do not use a misleading name: instead, a temporary version is set up as `\expl_pdfstrcmp`. The latter is then picked up and moved to the correct place by l3names. In earlier versions of the code, the pdftexcmds package was loaded to do this task. However, that raises some issues in "generic" (it fails with ConTeXt MkIV), and also adds a hardly-needed dependency.

Note that LuaTeX prior to version 0.36 is not supported by expl3: here that means simply skipping the definition, which will then be picked up later. This definition may need to be done twice: one "now" and once at the start of every job. The latter can occur in package mode if for example a custom format is being constructed. To achieve this while not requiring a separate file, the Lua code is saved into a macro then used twice.

```
\begingroup
```

```
  \expandafter\ifx\csname directlua\endcsname\relax
  \else
    \ifnum\luatexversion<36 %
    \else
      \catcode`\_=11 %
      \catcode`\:=11 %
      \def\tempa
        {%
          l3kernel = l3kernel or { }
          function l3kernel.strcmp (A, B)
            if A == B then
              tex.write ("0")
            elseif A < B then
              tex.write ("-1")
            else
              tex.write ("1")
            end
          end
        }
      \directlua{\tempa}
```

A test for LuaTEX in IniTEX mode.

```
      \ifnum 0%
        \directlua
          {%
            if status.ini_version then
              tex.write("1")
            end
          }>0 %
        \global\everyjob\expandafter
          {%
            \the\expandafter\everyjob
            \expandafter\luatex_directlua:D\expandafter{\tempa}%
          }
      \fi
      \long\gdef\expl_pdfstrcmp#1#2%
        {%
          \luatex_directlua:D
            {%
              l3kernel.strcmp
                (
                  "\luatex_luaescapestring:D {#1}" ,
                  "\luatex_luaescapestring:D {#2}"
                )
            }%
        }
    \fi
  \fi
\endgroup
```

(*End definition for* `\expl_pdfstrcmp`. *This function is documented on page* **??**.)

1.3 Engine requirements

The code currently requires functionality equivalent to `\pdfstrcmp` in addition to ε-TeX. This is picked up by testing for the `\pdfstrcmp` primitive or emulation as `\expl_pdfstrcmp`, as the ε-TeX primitives must be available if it is.

```
\begingroup
  \def\next{\endgroup}
  \def\ShortText{Required primitives not found}%
  \def\LongText%
    {%
      LaTeX3 requires the e-TeX primitives and \string\pdfstrcmp.\LineBreak
      \LineBreak
      These are available in engine versions:\LineBreak
      - pdfTeX 1.30\LineBreak
      - XeTeX 0.9994\LineBreak
      - LuaTeX 0.40\LineBreak
      or later.\LineBreak
      \LineBreak
    }%
  \expandafter\ifx\csname pdfstrcmp\endcsname\relax
    \catcode`\_=11 %
    \expandafter\ifx\csname expl_pdfstrcmp\endcsname\relax
⟨*initex⟩
      \def\LineBreak{^^J}%
      \edef\next
        {%
          \newlinechar`\noexpand\^^J\relax
          \errhelp
            {%
              \LongText
              For pdfTeX and XeTeX the '-etex' command-line switch is also
              needed.\LineBreak
              \LineBreak
              Format building will abort!\LineBreak
            }%
          \errmessage{\ShortText}%
          \endgroup
          \noexpand\end
        }%
⟨/initex⟩
⟨*package⟩
      \def\LineBreak{\noexpand\MessageBreak}%
      \expandafter\ifx\csname PackageError\endcsname\relax
        \def\LineBreak{^^J}%
        \begingroup
          \def\PackageError#1#2#3%
            {%
              \endgroup
              \errhelp{#3}%
              \errmessage{#1 Error: #2!}
```

```
          }%
      \fi
      \edef\next
        {%
          \noexpand\PackageError{expl3}{\ShortText}
            {\LongText Loading of expl3 will abort!}%
          \endgroup
          \noexpand\endinput
        }%
⟨/package⟩
    \fi
  \fi
\next
```

1.4 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 LaTeX only, load etex as otherwise we are likely to get into trouble with registers. 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.

```
⟨*package⟩
\begingroup
  \def\@tempa{LaTeX2e}
  \def\next{}
  \ifx\fmtname\@tempa
    \def\next
      {%
        \RequirePackage{etex}%
        \csname reserveinserts\endcsname{32}%
      }
  \fi
\expandafter\endgroup
\next
⟨/package⟩
```

If LuaTeX is in use there are more allocators to create. ConTeXt already does that, so skip any loading in that case: this is done using the `\newattribute` allocator. Otherwise, there is a need for a test to differentiate between LaTeX 2_ε and plain.

```
⟨*package⟩
\ifdefined\directlua
  \ifdefined\newattribute
  \else
    \ifdefined\RequirePackage
      \RequirePackage{luatex}
    \else
      \input luatex.sty %
    \fi
  \fi
\fi
⟨/package⟩
```

1.5 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.

```
\protected\def\ExplSyntaxOff{}
⟨*package⟩
\protected\edef\ExplSyntaxOff
  {%
    \protected\def\ExplSyntaxOff{}%
    \catcode   9 = \the\catcode   9\relax
    \catcode  32 = \the\catcode  32\relax
    \catcode  34 = \the\catcode  34\relax
    \catcode  38 = \the\catcode  38\relax
    \catcode  58 = \the\catcode  58\relax
    \catcode  94 = \the\catcode  94\relax
    \catcode  95 = \the\catcode  95\relax
    \catcode 124 = \the\catcode 124\relax
    \catcode 126 = \the\catcode 126\relax
    \endlinechar = \the\endlinechar\relax
    \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0\relax
  }
⟨/package⟩
```

(*End definition for* `\ExplSyntaxOff`. *This function is documented on page 6.*)

The code environment is now set up.

```
\catcode 9   = 9\relax
\catcode 32  = 9\relax
\catcode 34  = 12\relax
\catcode 58  = 11\relax
\catcode 94  = 7\relax
\catcode 95  = 11\relax
\catcode 124 = 12\relax
\catcode 126 = 10\relax
```

```
\endlinechar = 32\relax
```

\l__kernel_expl_bool

The status for experimental code syntax: this is on at present.

```
\chardef\l__kernel_expl_bool = 1 ~
```

(*End definition for* `\l__kernel_expl_bool`. *This variable is documented on page 7.*)

\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!

```
\protected \def \ExplSyntaxOn
  {
    \bool_if:NF \l__kernel_expl_bool
      {
        \cs_set_protected_nopar:Npx \ExplSyntaxOff
          {
            \char_set_catcode:nn { 9 }   { \char_value_catcode:n { 9 } }
            \char_set_catcode:nn { 32 }  { \char_value_catcode:n { 32 } }
            \char_set_catcode:nn { 34 }  { \char_value_catcode:n { 34 } }
            \char_set_catcode:nn { 38 }  { \char_value_catcode:n { 38 } }
            \char_set_catcode:nn { 58 }  { \char_value_catcode:n { 58 } }
            \char_set_catcode:nn { 94 }  { \char_value_catcode:n { 94 } }
            \char_set_catcode:nn { 95 }  { \char_value_catcode:n { 95 } }
            \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
            \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
            \tex_endlinechar:D =
              \tex_the:D \tex_endlinechar:D \scan_stop:
            \bool_set_false:N \l__kernel_expl_bool
            \cs_set_protected_nopar:Npn \ExplSyntaxOff { }
          }
      }
    \char_set_catcode_ignore:n           { 9 }   % tab
    \char_set_catcode_ignore:n           { 32 }  % space
    \char_set_catcode_other:n            { 34 }  % double quote
    \char_set_catcode_alignment:n        { 38 }  % ampersand
    \char_set_catcode_letter:n           { 58 }  % colon
    \char_set_catcode_math_superscript:n { 94 }  % circumflex
    \char_set_catcode_letter:n           { 95 }  % underscore
    \char_set_catcode_other:n            { 124 } % pipe
    \char_set_catcode_space:n            { 126 } % tilde
    \tex_endlinechar:D = 32 \scan_stop:
    \bool_set_true:N \l__kernel_expl_bool
  }
```

(*End definition for* `\ExplSyntaxOn`. *This function is documented on page 6.*)

```
⟨/initex | package⟩
```

2 l3names implementation

```
⟨*initex | package⟩
```

No prefix substitution here.

```
⟨@@=⟩
```

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`. *This function is documented on page* ??.)

The `\let` primitive is renamed by hand first as it is essential for the entire process to follow. This also uses `\global`, as that way we avoid leaving an unneeded csname in the hash table.

```
\let \tex_global:D \global
\let \tex_let:D    \let
```

Everything is inside a (rather long) group, which keeps `\__kernel_primitive:NN` trapped.

```
\begingroup
```

`\__kernel_primitive:NN` A temporary function to actually do the renaming. This also allows the original names to be removed in format mode.

```
  \long \def \__kernel_primitive:NN #1#2
    {
      \tex_global:D \tex_let:D #2 #1
⟨*initex⟩
      \tex_global:D \tex_let:D #1 \tex_undefined:D
⟨/initex⟩
    }
```

(*End definition for* `\__kernel_primitive:NN`.)

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.

```
  \__kernel_primitive:NN \                          \tex_space:D
  \__kernel_primitive:NN \/                         \tex_italiccorrection:D
  \__kernel_primitive:NN \-                         \tex_hyphen:D
```

Now all the other primitives.

```
  \__kernel_primitive:NN \let                       \tex_let:D
  \__kernel_primitive:NN \def                       \tex_def:D
  \__kernel_primitive:NN \edef                      \tex_edef:D
  \__kernel_primitive:NN \gdef                      \tex_gdef:D
  \__kernel_primitive:NN \xdef                      \tex_xdef:D
  \__kernel_primitive:NN \chardef                   \tex_chardef:D
  \__kernel_primitive:NN \countdef                  \tex_countdef:D
  \__kernel_primitive:NN \dimendef                  \tex_dimendef:D
  \__kernel_primitive:NN \skipdef                   \tex_skipdef:D
```

```
\__kernel_primitive:NN \muskipdef              \tex_muskipdef:D
\__kernel_primitive:NN \mathchardef            \tex_mathchardef:D
\__kernel_primitive:NN \toksdef                \tex_toksdef:D
\__kernel_primitive:NN \futurelet              \tex_futurelet:D
\__kernel_primitive:NN \advance                \tex_advance:D
\__kernel_primitive:NN \divide                 \tex_divide:D
\__kernel_primitive:NN \multiply               \tex_multiply:D
\__kernel_primitive:NN \font                   \tex_font:D
\__kernel_primitive:NN \fam                    \tex_fam:D
\__kernel_primitive:NN \global                 \tex_global:D
\__kernel_primitive:NN \long                   \tex_long:D
\__kernel_primitive:NN \outer                  \tex_outer:D
\__kernel_primitive:NN \setlanguage            \tex_setlanguage:D
\__kernel_primitive:NN \globaldefs             \tex_globaldefs:D
\__kernel_primitive:NN \afterassignment        \tex_afterassignment:D
\__kernel_primitive:NN \aftergroup             \tex_aftergroup:D
\__kernel_primitive:NN \expandafter            \tex_expandafter:D
\__kernel_primitive:NN \noexpand               \tex_noexpand:D
\__kernel_primitive:NN \begingroup             \tex_begingroup:D
\__kernel_primitive:NN \endgroup               \tex_endgroup:D
\__kernel_primitive:NN \halign                 \tex_halign:D
\__kernel_primitive:NN \valign                 \tex_valign:D
\__kernel_primitive:NN \cr                     \tex_cr:D
\__kernel_primitive:NN \crcr                   \tex_crcr:D
\__kernel_primitive:NN \noalign                \tex_noalign:D
\__kernel_primitive:NN \omit                   \tex_omit:D
\__kernel_primitive:NN \span                   \tex_span:D
\__kernel_primitive:NN \tabskip                \tex_tabskip:D
\__kernel_primitive:NN \everycr                \tex_everycr:D
\__kernel_primitive:NN \if                     \tex_if:D
\__kernel_primitive:NN \ifcase                 \tex_ifcase:D
\__kernel_primitive:NN \ifcat                  \tex_ifcat:D
\__kernel_primitive:NN \ifnum                  \tex_ifnum:D
\__kernel_primitive:NN \ifodd                  \tex_ifodd:D
\__kernel_primitive:NN \ifdim                  \tex_ifdim:D
\__kernel_primitive:NN \ifeof                  \tex_ifeof:D
\__kernel_primitive:NN \ifhbox                 \tex_ifhbox:D
\__kernel_primitive:NN \ifvbox                 \tex_ifvbox:D
\__kernel_primitive:NN \ifvoid                 \tex_ifvoid:D
\__kernel_primitive:NN \ifx                    \tex_ifx:D
\__kernel_primitive:NN \iffalse                \tex_iffalse:D
\__kernel_primitive:NN \iftrue                 \tex_iftrue:D
\__kernel_primitive:NN \ifhmode                \tex_ifhmode:D
\__kernel_primitive:NN \ifmmode                \tex_ifmmode:D
\__kernel_primitive:NN \ifvmode                \tex_ifvmode:D
\__kernel_primitive:NN \ifinner                \tex_ifinner:D
\__kernel_primitive:NN \else                   \tex_else:D
\__kernel_primitive:NN \fi                     \tex_fi:D
\__kernel_primitive:NN \or                     \tex_or:D
\__kernel_primitive:NN \immediate              \tex_immediate:D
```

```
\__kernel_primitive:NN \closeout            \tex_closeout:D
\__kernel_primitive:NN \openin              \tex_openin:D
\__kernel_primitive:NN \openout             \tex_openout:D
\__kernel_primitive:NN \read                \tex_read:D
\__kernel_primitive:NN \write               \tex_write:D
\__kernel_primitive:NN \closein             \tex_closein:D
\__kernel_primitive:NN \newlinechar         \tex_newlinechar:D
\__kernel_primitive:NN \input               \tex_input:D
\__kernel_primitive:NN \endinput            \tex_endinput:D
\__kernel_primitive:NN \inputlineno         \tex_inputlineno:D
\__kernel_primitive:NN \errmessage          \tex_errmessage:D
\__kernel_primitive:NN \message             \tex_message:D
\__kernel_primitive:NN \show                \tex_show:D
\__kernel_primitive:NN \showthe             \tex_showthe:D
\__kernel_primitive:NN \showbox             \tex_showbox:D
\__kernel_primitive:NN \showlists           \tex_showlists:D
\__kernel_primitive:NN \errhelp             \tex_errhelp:D
\__kernel_primitive:NN \errorcontextlines   \tex_errorcontextlines:D
\__kernel_primitive:NN \tracingcommands     \tex_tracingcommands:D
\__kernel_primitive:NN \tracinglostchars    \tex_tracinglostchars:D
\__kernel_primitive:NN \tracingmacros       \tex_tracingmacros:D
\__kernel_primitive:NN \tracingonline       \tex_tracingonline:D
\__kernel_primitive:NN \tracingoutput       \tex_tracingoutput:D
\__kernel_primitive:NN \tracingpages        \tex_tracingpages:D
\__kernel_primitive:NN \tracingparagraphs   \tex_tracingparagraphs:D
\__kernel_primitive:NN \tracingrestores     \tex_tracingrestores:D
\__kernel_primitive:NN \tracingstats        \tex_tracingstats:D
\__kernel_primitive:NN \pausing             \tex_pausing:D
\__kernel_primitive:NN \showboxbreadth      \tex_showboxbreadth:D
\__kernel_primitive:NN \showboxdepth        \tex_showboxdepth:D
\__kernel_primitive:NN \batchmode           \tex_batchmode:D
\__kernel_primitive:NN \errorstopmode       \tex_errorstopmode:D
\__kernel_primitive:NN \nonstopmode         \tex_nonstopmode:D
\__kernel_primitive:NN \scrollmode          \tex_scrollmode:D
\__kernel_primitive:NN \end                 \tex_end:D
\__kernel_primitive:NN \csname              \tex_csname:D
\__kernel_primitive:NN \endcsname           \tex_endcsname:D
\__kernel_primitive:NN \ignorespaces        \tex_ignorespaces:D
\__kernel_primitive:NN \relax               \tex_relax:D
\__kernel_primitive:NN \the                 \tex_the:D
\__kernel_primitive:NN \mag                 \tex_mag:D
\__kernel_primitive:NN \language            \tex_language:D
\__kernel_primitive:NN \mark                \tex_mark:D
\__kernel_primitive:NN \topmark             \tex_topmark:D
\__kernel_primitive:NN \firstmark           \tex_firstmark:D
\__kernel_primitive:NN \botmark             \tex_botmark:D
\__kernel_primitive:NN \splitfirstmark      \tex_splitfirstmark:D
\__kernel_primitive:NN \splitbotmark        \tex_splitbotmark:D
\__kernel_primitive:NN \fontname            \tex_fontname:D
\__kernel_primitive:NN \escapechar          \tex_escapechar:D
```

```
\__kernel_primitive:NN \endlinechar           \tex_endlinechar:D
\__kernel_primitive:NN \mathchoice            \tex_mathchoice:D
\__kernel_primitive:NN \delimiter             \tex_delimiter:D
\__kernel_primitive:NN \mathaccent            \tex_mathaccent:D
\__kernel_primitive:NN \mathchar              \tex_mathchar:D
\__kernel_primitive:NN \mskip                 \tex_mskip:D
\__kernel_primitive:NN \radical               \tex_radical:D
\__kernel_primitive:NN \vcenter               \tex_vcenter:D
\__kernel_primitive:NN \mkern                 \tex_mkern:D
\__kernel_primitive:NN \above                 \tex_above:D
\__kernel_primitive:NN \abovewithdelims       \tex_abovewithdelims:D
\__kernel_primitive:NN \atop                  \tex_atop:D
\__kernel_primitive:NN \atopwithdelims        \tex_atopwithdelims:D
\__kernel_primitive:NN \over                  \tex_over:D
\__kernel_primitive:NN \overwithdelims        \tex_overwithdelims:D
\__kernel_primitive:NN \displaystyle          \tex_displaystyle:D
\__kernel_primitive:NN \textstyle             \tex_textstyle:D
\__kernel_primitive:NN \scriptstyle           \tex_scriptstyle:D
\__kernel_primitive:NN \scriptscriptstyle     \tex_scriptscriptstyle:D
\__kernel_primitive:NN \nonscript             \tex_nonscript:D
\__kernel_primitive:NN \eqno                  \tex_eqno:D
\__kernel_primitive:NN \leqno                 \tex_leqno:D
\__kernel_primitive:NN \abovedisplayshortskip \tex_abovedisplayshortskip:D
\__kernel_primitive:NN \abovedisplayskip      \tex_abovedisplayskip:D
\__kernel_primitive:NN \belowdisplayshortskip \tex_belowdisplayshortskip:D
\__kernel_primitive:NN \belowdisplayskip      \tex_belowdisplayskip:D
\__kernel_primitive:NN \displaywidowpenalty   \tex_displaywidowpenalty:D
\__kernel_primitive:NN \displayindent         \tex_displayindent:D
\__kernel_primitive:NN \displaywidth          \tex_displaywidth:D
\__kernel_primitive:NN \everydisplay          \tex_everydisplay:D
\__kernel_primitive:NN \predisplaysize        \tex_predisplaysize:D
\__kernel_primitive:NN \predisplaypenalty     \tex_predisplaypenalty:D
\__kernel_primitive:NN \postdisplaypenalty    \tex_postdisplaypenalty:D
\__kernel_primitive:NN \mathbin               \tex_mathbin:D
\__kernel_primitive:NN \mathclose             \tex_mathclose:D
\__kernel_primitive:NN \mathinner             \tex_mathinner:D
\__kernel_primitive:NN \mathop                \tex_mathop:D
\__kernel_primitive:NN \displaylimits         \tex_displaylimits:D
\__kernel_primitive:NN \limits                \tex_limits:D
\__kernel_primitive:NN \nolimits              \tex_nolimits:D
\__kernel_primitive:NN \mathopen              \tex_mathopen:D
\__kernel_primitive:NN \mathord               \tex_mathord:D
\__kernel_primitive:NN \mathpunct             \tex_mathpunct:D
\__kernel_primitive:NN \mathrel               \tex_mathrel:D
\__kernel_primitive:NN \overline              \tex_overline:D
\__kernel_primitive:NN \underline             \tex_underline:D
\__kernel_primitive:NN \left                  \tex_left:D
\__kernel_primitive:NN \right                 \tex_right:D
\__kernel_primitive:NN \binoppenalty          \tex_binoppenalty:D
\__kernel_primitive:NN \relpenalty            \tex_relpenalty:D
```

```
\__kernel_primitive:NN \delimitershortfall    \tex_delimitershortfall:D
\__kernel_primitive:NN \delimiterfactor       \tex_delimiterfactor:D
\__kernel_primitive:NN \nulldelimiterspace    \tex_nulldelimiterspace:D
\__kernel_primitive:NN \everymath             \tex_everymath:D
\__kernel_primitive:NN \mathsurround          \tex_mathsurround:D
\__kernel_primitive:NN \medmuskip             \tex_medmuskip:D
\__kernel_primitive:NN \thinmuskip            \tex_thinmuskip:D
\__kernel_primitive:NN \thickmuskip           \tex_thickmuskip:D
\__kernel_primitive:NN \scriptspace           \tex_scriptspace:D
\__kernel_primitive:NN \noboundary            \tex_noboundary:D
\__kernel_primitive:NN \accent                \tex_accent:D
\__kernel_primitive:NN \char                  \tex_char:D
\__kernel_primitive:NN \discretionary         \tex_discretionary:D
\__kernel_primitive:NN \hfil                  \tex_hfil:D
\__kernel_primitive:NN \hfilneg               \tex_hfilneg:D
\__kernel_primitive:NN \hfill                 \tex_hfill:D
\__kernel_primitive:NN \hskip                 \tex_hskip:D
\__kernel_primitive:NN \hss                   \tex_hss:D
\__kernel_primitive:NN \vfil                  \tex_vfil:D
\__kernel_primitive:NN \vfilneg               \tex_vfilneg:D
\__kernel_primitive:NN \vfill                 \tex_vfill:D
\__kernel_primitive:NN \vskip                 \tex_vskip:D
\__kernel_primitive:NN \vss                   \tex_vss:D
\__kernel_primitive:NN \unskip                \tex_unskip:D
\__kernel_primitive:NN \kern                  \tex_kern:D
\__kernel_primitive:NN \unkern                \tex_unkern:D
\__kernel_primitive:NN \hrule                 \tex_hrule:D
\__kernel_primitive:NN \vrule                 \tex_vrule:D
\__kernel_primitive:NN \leaders               \tex_leaders:D
\__kernel_primitive:NN \cleaders              \tex_cleaders:D
\__kernel_primitive:NN \xleaders              \tex_xleaders:D
\__kernel_primitive:NN \lastkern              \tex_lastkern:D
\__kernel_primitive:NN \lastskip              \tex_lastskip:D
\__kernel_primitive:NN \indent                \tex_indent:D
\__kernel_primitive:NN \par                   \tex_par:D
\__kernel_primitive:NN \noindent              \tex_noindent:D
\__kernel_primitive:NN \vadjust               \tex_vadjust:D
\__kernel_primitive:NN \baselineskip          \tex_baselineskip:D
\__kernel_primitive:NN \lineskip              \tex_lineskip:D
\__kernel_primitive:NN \lineskiplimit         \tex_lineskiplimit:D
\__kernel_primitive:NN \clubpenalty           \tex_clubpenalty:D
\__kernel_primitive:NN \widowpenalty          \tex_widowpenalty:D
\__kernel_primitive:NN \exhyphenpenalty       \tex_exhyphenpenalty:D
\__kernel_primitive:NN \hyphenpenalty         \tex_hyphenpenalty:D
\__kernel_primitive:NN \linepenalty           \tex_linepenalty:D
\__kernel_primitive:NN \doublehyphendemerits  \tex_doublehyphendemerits:D
\__kernel_primitive:NN \finalhyphendemerits   \tex_finalhyphendemerits:D
\__kernel_primitive:NN \adjdemerits           \tex_adjdemerits:D
\__kernel_primitive:NN \hangafter             \tex_hangafter:D
\__kernel_primitive:NN \hangindent            \tex_hangindent:D
```

```
\__kernel_primitive:NN \parshape               \tex_parshape:D
\__kernel_primitive:NN \hsize                  \tex_hsize:D
\__kernel_primitive:NN \lefthyphenmin          \tex_lefthyphenmin:D
\__kernel_primitive:NN \righthyphenmin         \tex_righthyphenmin:D
\__kernel_primitive:NN \leftskip               \tex_leftskip:D
\__kernel_primitive:NN \rightskip              \tex_rightskip:D
\__kernel_primitive:NN \looseness              \tex_looseness:D
\__kernel_primitive:NN \parskip                \tex_parskip:D
\__kernel_primitive:NN \parindent              \tex_parindent:D
\__kernel_primitive:NN \uchyph                 \tex_uchyph:D
\__kernel_primitive:NN \emergencystretch       \tex_emergencystretch:D
\__kernel_primitive:NN \pretolerance           \tex_pretolerance:D
\__kernel_primitive:NN \tolerance              \tex_tolerance:D
\__kernel_primitive:NN \spaceskip              \tex_spaceskip:D
\__kernel_primitive:NN \xspaceskip             \tex_xspaceskip:D
\__kernel_primitive:NN \parfillskip            \tex_parfillskip:D
\__kernel_primitive:NN \everypar               \tex_everypar:D
\__kernel_primitive:NN \prevgraf               \tex_prevgraf:D
\__kernel_primitive:NN \spacefactor            \tex_spacefactor:D
\__kernel_primitive:NN \shipout                \tex_shipout:D
\__kernel_primitive:NN \vsize                  \tex_vsize:D
\__kernel_primitive:NN \interlinepenalty       \tex_interlinepenalty:D
\__kernel_primitive:NN \brokenpenalty          \tex_brokenpenalty:D
\__kernel_primitive:NN \topskip                \tex_topskip:D
\__kernel_primitive:NN \maxdeadcycles          \tex_maxdeadcycles:D
\__kernel_primitive:NN \maxdepth               \tex_maxdepth:D
\__kernel_primitive:NN \output                 \tex_output:D
\__kernel_primitive:NN \deadcycles             \tex_deadcycles:D
\__kernel_primitive:NN \pagedepth              \tex_pagedepth:D
\__kernel_primitive:NN \pagestretch            \tex_pagestretch:D
\__kernel_primitive:NN \pagefilstretch         \tex_pagefilstretch:D
\__kernel_primitive:NN \pagefillstretch        \tex_pagefillstretch:D
\__kernel_primitive:NN \pagefilllstretch       \tex_pagefilllstretch:D
\__kernel_primitive:NN \pageshrink             \tex_pageshrink:D
\__kernel_primitive:NN \pagegoal               \tex_pagegoal:D
\__kernel_primitive:NN \pagetotal              \tex_pagetotal:D
\__kernel_primitive:NN \outputpenalty          \tex_outputpenalty:D
\__kernel_primitive:NN \hoffset                \tex_hoffset:D
\__kernel_primitive:NN \voffset                \tex_voffset:D
\__kernel_primitive:NN \insert                 \tex_insert:D
\__kernel_primitive:NN \holdinginserts         \tex_holdinginserts:D
\__kernel_primitive:NN \floatingpenalty        \tex_floatingpenalty:D
\__kernel_primitive:NN \insertpenalties        \tex_insertpenalties:D
\__kernel_primitive:NN \lower                  \tex_lower:D
\__kernel_primitive:NN \moveleft               \tex_moveleft:D
\__kernel_primitive:NN \moveright              \tex_moveright:D
\__kernel_primitive:NN \raise                  \tex_raise:D
\__kernel_primitive:NN \copy                   \tex_copy:D
\__kernel_primitive:NN \lastbox                \tex_lastbox:D
\__kernel_primitive:NN \vsplit                 \tex_vsplit:D
```

```
\__kernel_primitive:NN \unhbox              \tex_unhbox:D
\__kernel_primitive:NN \unhcopy             \tex_unhcopy:D
\__kernel_primitive:NN \unvbox              \tex_unvbox:D
\__kernel_primitive:NN \unvcopy             \tex_unvcopy:D
\__kernel_primitive:NN \setbox              \tex_setbox:D
\__kernel_primitive:NN \hbox                \tex_hbox:D
\__kernel_primitive:NN \vbox                \tex_vbox:D
\__kernel_primitive:NN \vtop                \tex_vtop:D
\__kernel_primitive:NN \prevdepth           \tex_prevdepth:D
\__kernel_primitive:NN \badness             \tex_badness:D
\__kernel_primitive:NN \hbadness            \tex_hbadness:D
\__kernel_primitive:NN \vbadness            \tex_vbadness:D
\__kernel_primitive:NN \hfuzz               \tex_hfuzz:D
\__kernel_primitive:NN \vfuzz               \tex_vfuzz:D
\__kernel_primitive:NN \overfullrule        \tex_overfullrule:D
\__kernel_primitive:NN \boxmaxdepth         \tex_boxmaxdepth:D
\__kernel_primitive:NN \splitmaxdepth       \tex_splitmaxdepth:D
\__kernel_primitive:NN \splittopskip        \tex_splittopskip:D
\__kernel_primitive:NN \everyhbox           \tex_everyhbox:D
\__kernel_primitive:NN \everyvbox           \tex_everyvbox:D
\__kernel_primitive:NN \nullfont            \tex_nullfont:D
\__kernel_primitive:NN \textfont            \tex_textfont:D
\__kernel_primitive:NN \scriptfont          \tex_scriptfont:D
\__kernel_primitive:NN \scriptscriptfont    \tex_scriptscriptfont:D
\__kernel_primitive:NN \fontdimen           \tex_fontdimen:D
\__kernel_primitive:NN \hyphenchar          \tex_hyphenchar:D
\__kernel_primitive:NN \skewchar            \tex_skewchar:D
\__kernel_primitive:NN \defaulthyphenchar   \tex_defaulthyphenchar:D
\__kernel_primitive:NN \defaultskewchar     \tex_defaultskewchar:D
\__kernel_primitive:NN \number              \tex_number:D
\__kernel_primitive:NN \romannumeral        \tex_romannumeral:D
\__kernel_primitive:NN \string              \tex_string:D
\__kernel_primitive:NN \lowercase           \tex_lowercase:D
\__kernel_primitive:NN \uppercase           \tex_uppercase:D
\__kernel_primitive:NN \meaning             \tex_meaning:D
\__kernel_primitive:NN \penalty             \tex_penalty:D
\__kernel_primitive:NN \unpenalty           \tex_unpenalty:D
\__kernel_primitive:NN \lastpenalty         \tex_lastpenalty:D
\__kernel_primitive:NN \special             \tex_special:D
\__kernel_primitive:NN \dump                \tex_dump:D
\__kernel_primitive:NN \patterns            \tex_patterns:D
\__kernel_primitive:NN \hyphenation         \tex_hyphenation:D
\__kernel_primitive:NN \time                \tex_time:D
\__kernel_primitive:NN \day                 \tex_day:D
\__kernel_primitive:NN \month               \tex_month:D
\__kernel_primitive:NN \year                \tex_year:D
\__kernel_primitive:NN \jobname             \tex_jobname:D
\__kernel_primitive:NN \everyjob            \tex_everyjob:D
\__kernel_primitive:NN \count               \tex_count:D
\__kernel_primitive:NN \dimen               \tex_dimen:D
```

```
\__kernel_primitive:NN \skip                \tex_skip:D
\__kernel_primitive:NN \toks                \tex_toks:D
\__kernel_primitive:NN \muskip              \tex_muskip:D
\__kernel_primitive:NN \box                 \tex_box:D
\__kernel_primitive:NN \wd                  \tex_wd:D
\__kernel_primitive:NN \ht                  \tex_ht:D
\__kernel_primitive:NN \dp                  \tex_dp:D
\__kernel_primitive:NN \catcode             \tex_catcode:D
\__kernel_primitive:NN \delcode             \tex_delcode:D
\__kernel_primitive:NN \sfcode              \tex_sfcode:D
\__kernel_primitive:NN \lccode              \tex_lccode:D
\__kernel_primitive:NN \uccode              \tex_uccode:D
\__kernel_primitive:NN \mathcode            \tex_mathcode:D
```

Since LaTeX3 requires at least the ε-TeX extensions, we also rename the additional primitives. These are all given the prefix `\etex_`.

```
\__kernel_primitive:NN \ifdefined           \etex_ifdefined:D
\__kernel_primitive:NN \ifcsname            \etex_ifcsname:D
\__kernel_primitive:NN \unless              \etex_unless:D
\__kernel_primitive:NN \eTeXversion         \etex_eTeXversion:D
\__kernel_primitive:NN \eTeXrevision        \etex_eTeXrevision:D
\__kernel_primitive:NN \marks               \etex_marks:D
\__kernel_primitive:NN \topmarks            \etex_topmarks:D
\__kernel_primitive:NN \firstmarks          \etex_firstmarks:D
\__kernel_primitive:NN \botmarks            \etex_botmarks:D
\__kernel_primitive:NN \splitfirstmarks     \etex_splitfirstmarks:D
\__kernel_primitive:NN \splitbotmarks       \etex_splitbotmarks:D
\__kernel_primitive:NN \unexpanded          \etex_unexpanded:D
\__kernel_primitive:NN \detokenize          \etex_detokenize:D
\__kernel_primitive:NN \scantokens          \etex_scantokens:D
\__kernel_primitive:NN \showtokens          \etex_showtokens:D
\__kernel_primitive:NN \readline            \etex_readline:D
\__kernel_primitive:NN \tracingassigns      \etex_tracingassigns:D
\__kernel_primitive:NN \tracingscantokens   \etex_tracingscantokens:D
\__kernel_primitive:NN \tracingnesting      \etex_tracingnesting:D
\__kernel_primitive:NN \tracingifs          \etex_tracingifs:D
\__kernel_primitive:NN \currentiflevel      \etex_currentiflevel:D
\__kernel_primitive:NN \currentifbranch     \etex_currentifbranch:D
\__kernel_primitive:NN \currentiftype       \etex_currentiftype:D
\__kernel_primitive:NN \tracinggroups       \etex_tracinggroups:D
\__kernel_primitive:NN \currentgrouplevel   \etex_currentgrouplevel:D
\__kernel_primitive:NN \currentgrouptype    \etex_currentgrouptype:D
\__kernel_primitive:NN \showgroups          \etex_showgroups:D
\__kernel_primitive:NN \showifs             \etex_showifs:D
\__kernel_primitive:NN \interactionmode     \etex_interactionmode:D
\__kernel_primitive:NN \lastnodetype        \etex_lastnodetype:D
\__kernel_primitive:NN \iffontchar          \etex_iffontchar:D
\__kernel_primitive:NN \fontcharht          \etex_fontcharht:D
\__kernel_primitive:NN \fontchardp          \etex_fontchardp:D
\__kernel_primitive:NN \fontcharwd          \etex_fontcharwd:D
```

```
\__kernel_primitive:NN \fontcharic            \etex_fontcharic:D
\__kernel_primitive:NN \parshapeindent        \etex_parshapeindent:D
\__kernel_primitive:NN \parshapelength        \etex_parshapelength:D
\__kernel_primitive:NN \parshapedimen         \etex_parshapedimen:D
\__kernel_primitive:NN \numexpr               \etex_numexpr:D
\__kernel_primitive:NN \dimexpr               \etex_dimexpr:D
\__kernel_primitive:NN \glueexpr              \etex_glueexpr:D
\__kernel_primitive:NN \muexpr                \etex_muexpr:D
\__kernel_primitive:NN \gluestretch           \etex_gluestretch:D
\__kernel_primitive:NN \glueshrink            \etex_glueshrink:D
\__kernel_primitive:NN \gluestretchorder      \etex_gluestretchorder:D
\__kernel_primitive:NN \glueshrinkorder       \etex_glueshrinkorder:D
\__kernel_primitive:NN \gluetomu              \etex_gluetomu:D
\__kernel_primitive:NN \mutoglue              \etex_mutoglue:D
\__kernel_primitive:NN \lastlinefit           \etex_lastlinefit:D
\__kernel_primitive:NN \interlinepenalties    \etex_interlinepenalties:D
\__kernel_primitive:NN \clubpenalties         \etex_clubpenalties:D
\__kernel_primitive:NN \widowpenalties        \etex_widowpenalties:D
\__kernel_primitive:NN \displaywidowpenalties \etex_displaywidowpenalties:D
\__kernel_primitive:NN \middle                \etex_middle:D
\__kernel_primitive:NN \savinghyphcodes       \etex_savinghyphcodes:D
\__kernel_primitive:NN \savingvdiscards       \etex_savingvdiscards:D
\__kernel_primitive:NN \pagediscards          \etex_pagediscards:D
\__kernel_primitive:NN \splitdiscards         \etex_splitdiscards:D
\__kernel_primitive:NN \TeXXeTstate           \etex_TeXXeTstate:D
\__kernel_primitive:NN \beginL                \etex_beginL:D
\__kernel_primitive:NN \endL                  \etex_endL:D
\__kernel_primitive:NN \beginR                \etex_beginR:D
\__kernel_primitive:NN \endR                  \etex_endR:D
\__kernel_primitive:NN \predisplaydirection   \etex_predisplaydirection:D
\__kernel_primitive:NN \everyeof              \etex_everyeof:D
\__kernel_primitive:NN \protected             \etex_protected: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. 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.

```
\__kernel_primitive:NN \pdfcreationdate       \pdftex_pdfcreationdate:D
\__kernel_primitive:NN \pdfcolorstack         \pdftex_pdfcolorstack:D
\__kernel_primitive:NN \pdfcompresslevel      \pdftex_pdfcompresslevel:D
\__kernel_primitive:NN \pdfdecimaldigits      \pdftex_pdfdecimaldigits:D
\__kernel_primitive:NN \pdfhorigin            \pdftex_pdfhorigin:D
\__kernel_primitive:NN \pdfinfo               \pdftex_pdfinfo:D
\__kernel_primitive:NN \pdflastxform          \pdftex_pdflastxform:D
\__kernel_primitive:NN \pdfliteral            \pdftex_pdfliteral:D
\__kernel_primitive:NN \pdfminorversion       \pdftex_pdfminorversion:D
\__kernel_primitive:NN \pdfobjcompresslevel   \pdftex_pdfobjcompresslevel:D
\__kernel_primitive:NN \pdfoutput             \pdftex_pdfoutput:D
\__kernel_primitive:NN \pdfrefxform           \pdftex_pdfrefxform:D
```

```
  \__kernel_primitive:NN \pdfrestore            \pdftex_pdfrestore:D
  \__kernel_primitive:NN \pdfsave               \pdftex_pdfsave:D
  \__kernel_primitive:NN \pdfsetmatrix          \pdftex_pdfsetmatrix:D
  \__kernel_primitive:NN \pdfpkresolution       \pdftex_pdfpkresolution:D
  \__kernel_primitive:NN \pdftexrevision        \pdftex_pdftextrevision:D
  \__kernel_primitive:NN \pdfvorigin            \pdftex_pdfvorigin:D
  \__kernel_primitive:NN \pdfxform              \pdftex_pdfxform:D
```

While these are not.

```
  \__kernel_primitive:NN \pdfstrcmp             \pdftex_strcmp:D
```

XƎTEX-specific primitives. Note that XƎTEX's `\strcmp` is handled earlier and is "rolled up" into `\pdfstrcmp`.

```
  \__kernel_primitive:NN \XeTeXversion          \xetex_XeTeXversion:D
```

Primitives from LuaTEX.

```
  \__kernel_primitive:NN \catcodetable          \luatex_catcodetable:D
  \__kernel_primitive:NN \directlua             \luatex_directlua:D
  \__kernel_primitive:NN \initcatcodetable      \luatex_initcatcodetable:D
  \__kernel_primitive:NN \latelua               \luatex_latelua:D
  \__kernel_primitive:NN \luaescapestring       \luatex_luaescapestring:D
  \__kernel_primitive:NN \luatexversion         \luatex_luatexversion:D
  \__kernel_primitive:NN \savecatcodetable      \luatex_savecatcodetable:D
  \__kernel_primitive:NN \Uchar                 \luatex_Uchar:D
```

Slightly more awkward are the directional primitives in LuaTEX. These come from Omega *via* Aleph, but we do not support those engines and so it seems most sensible to treat them as LuaTEX primitives for prefix purposes.

```
  \__kernel_primitive:NN \bodydir               \luatex_bodydir:D
  \__kernel_primitive:NN \mathdir               \luatex_mathdir:D
  \__kernel_primitive:NN \pagedir               \luatex_pagedir:D
  \__kernel_primitive:NN \pardir                \luatex_pardir:D
  \__kernel_primitive:NN \textdir               \luatex_textdir:D
```

The job is done: close the group (using the primitive renamed!).

```
\tex_endgroup:D
```

If the underlying engine is LuaTEX then the `\pdfstrcmp` primitive is emulated as `\expl_pdfstrcmp` as part of the bootstrap process. That can be detected as `\pdftex_strcmp:D` will not be defined: is that is the case, copy the code and then remove the temporary emulation.

```
\etex_ifdefined:D \pdftex_strcmp:D
\tex_else:D
  \tex_let:D \pdftex_strcmp:D \expl_pdfstrcmp
  \tex_let:D \expl_pdfstrcmp  \tex_undefined:D
\tex_fi:D
```

LATEX 2_ε will have moved a few primitives, so these are sorted out. A convenient test for LATEX 2_ε is the `\@@end` saved primitive.

```
⟨*package⟩
\etex_ifdefined:D \@@end
  \tex_let:D \tex_end:D               \@@end
```

```
  \tex_let:D \tex_everydisplay:D      \frozen@everydisplay
  \tex_let:D \tex_everymath:D         \frozen@everymath
  \tex_let:D \tex_hyphen:D            \@@hyph
  \tex_let:D \tex_input:D             \@@input
  \tex_let:D \tex_italiccorrection:D  \@@italiccorr
  \tex_let:D \tex_underline:D         \@@underline
```

That is also true for the LuaTEX primitives under LATEX 2ε.

```
  \tex_let:D \luatex_catcodetable:D      \luatexcatcodetable
  \tex_let:D \luatex_initcatcodetable:D  \luatexinitcatcodetable
  \tex_let:D \luatex_latelua:D           \luatexlatelua
  \tex_let:D \luatex_luaescapestring:D   \luatexluaescapestring
  \tex_let:D \luatex_savecatcodetable:D  \luatexsavecatcodetable
  \tex_let:D \luatex_Uchar:D             \luatexUchar
```

Which also covers those slightly odd ones.

```
  \tex_let:D \luatex_bodydir:D           \luatexbodydir
  \tex_let:D \luatex_mathdir:D           \luatexmathdir
  \tex_let:D \luatex_pagedir:D           \luatexpagedir
  \tex_let:D \luatex_pardir:D            \luatexpardir
  \tex_let:D \luatex_textdir:D           \luatextextdir
\tex_fi:D
```

For ConTEXt, two tests are needed. Both Mark II and Mark IV move several primitives: these are all covered by the first test, again using `\end` as a marker. For Mark IV, a few more primitives are moved: they are implemented using some Lua code in the current ConTEXt.

```
\etex_ifdefined:D \normalend
  \tex_let:D \tex_outer:D       \normalouter
  \tex_let:D \tex_input:D       \normalinput
  \tex_let:D \tex_end:D         \normalend
  \tex_let:D \tex_language:D    \normallanguage
  \tex_let:D \tex_vcenter:D     \normalvcneter
  \tex_let:D \tex_over:D        \normalover
  \tex_let:D \tex_mathop:D      \normalmathop
  \tex_let:D \tex_month:D       \normalmonth
  \tex_let:D \tex_everyjob:D    \normaleveryjob
  \tex_let:D \etex_unexpanded:D \normalunexpanded
\tex_fi:D
\etex_ifdefined:D \normalitaliccorrection
  \tex_let:D \tex_hoffset:D          \normalhoffset
  \tex_let:D \tex_italiccorrection:D \normalitaliccorrection
  \tex_let:D \tex_voffset:D          \normalvoffset
  \tex_let:D \etex_showtokens:D      \normalshowtokens
  \tex_let:D \luatex_bodydir:D       \spac_directions_normal_body_dir
  \tex_let:D \luatex_pagedir:D       \spac_directions_normal_page_dir
\tex_fi:D
\etex_ifdefined:D \normalleft
  \tex_let:D \tex_left:D   \normalleft
  \tex_let:D \tex_middle:D \normalmiddle
  \tex_let:D \tex_right:D  \normalright
```

```
\tex_fi:D
⟨/package⟩
⟨/initex | package⟩
```

3 l3basics implementation

```
⟨*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.[2]

\if_true: \if_false: \or: \else: \fi: \reverse_if:N \if:w \if_charcode:w \if_catcode:w \if_meaning:w

Then some conditionals.

```
\tex_let:D \if_true:          \tex_iftrue:D
\tex_let:D \if_false:         \tex_iffalse:D
\tex_let:D \or:               \tex_or:D
\tex_let:D \else:             \tex_else:D
\tex_let:D \fi:               \tex_fi:D
\tex_let:D \reverse_if:N      \etex_unless:D
\tex_let:D \if:w              \tex_if:D
\tex_let:D \if_charcode:w     \tex_if:D
\tex_let:D \if_catcode:w      \tex_ifcat:D
\tex_let:D \if_meaning:w      \tex_ifx:D
```

(*End definition for `\if_true:` and others. These functions are documented on page 24.*)

\if_mode_math: \if_mode_horizontal: \if_mode_vertical: \if_mode_inner:

TEX lets us detect some if its modes.

```
\tex_let:D \if_mode_math:       \tex_ifmmode:D
\tex_let:D \if_mode_horizontal: \tex_ifhmode:D
\tex_let:D \if_mode_vertical:   \tex_ifvmode:D
\tex_let:D \if_mode_inner:      \tex_ifinner:D
```

(*End definition for `\if_mode_math:` and others. These functions are documented on page 24.*)

\if_cs_exist:N \if_cs_exist:w \cs:w \cs_end:

Building csnames and testing if control sequences exist.

```
\tex_let:D \if_cs_exist:N     \etex_ifdefined:D
\tex_let:D \if_cs_exist:w     \etex_ifcsname:D
\tex_let:D \cs:w              \tex_csname:D
\tex_let:D \cs_end:           \tex_endcsname:D
```

(*End definition for `\if_cs_exist:N` and others. These functions are documented on page 17.*)

\exp_after:wN \exp_not:N \exp_not:n

The three `\exp_` functions are used in the l3expan module where they are described.

```
\tex_let:D \exp_after:wN      \tex_expandafter:D
\tex_let:D \exp_not:N         \tex_noexpand:D
\tex_let:D \exp_not:n         \etex_unexpanded:D
```

[2]This renaming gets expensive in terms of csname usage, an alternative scheme would be to just use the `\tex...:D` name in the cases where no good alternative exists.

(*End definition for* \exp_after:wN*,* \exp_not:N*, and* \exp_not:n*. These functions are documented on page 33.*)

\token_to_meaning:N
\token_to_str:N
\cs_meaning:N

Examining a control sequence or token.

```
\tex_let:D \token_to_meaning:N \tex_meaning:D
\tex_let:D \token_to_str:N     \tex_string:D
\tex_let:D \cs_meaning:N       \tex_meaning:D
```

(*End definition for* \token_to_meaning:N*,* \token_to_str:N*, and* \cs_meaning:N*. These functions are documented on page 16.*)

\scan_stop:
\group_begin:
\group_end:

The next three are basic functions for which there also exist versions that are safe inside alignments. These safe versions are defined in the l3prg module.

```
\tex_let:D \scan_stop:          \tex_relax:D
\tex_let:D \group_begin:        \tex_begingroup:D
\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.

```
\tex_let:D \if_int_compare:w   \tex_ifnum:D
\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 74.*)

\group_insert_after:N

Adding material after the end of a group.

```
\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.

```
\tex_long:D \tex_def:D \exp_args:Nc #1#2
  { \exp_after:wN #1 \cs:w #2 \cs_end: }
\tex_long:D \tex_def:D \exp_args:cc #1#2
  { \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* **??***.*)

\token_to_meaning:c
\token_to_str:c
\cs_meaning:c

A small number of variants defined by hand. Some of the necessary functions (\use_i:nn, \use_ii:nn, and \exp_args:NNc) are not defined at that point yet, but will be defined before those variants are used. The \cs_meaning:c command must check for an undefined control sequence to avoid defining it mistakenly.

```
\tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
\tex_long:D \tex_def:D \cs_meaning:c #1
  {
    \if_cs_exist:w #1 \cs_end:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
    { \exp_args:Nc \cs_meaning:N {#1} }
```

```
    { \tl_to_str:n {undefined} }
  }
\tex_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* ??.)

3.2 Defining some constants

`\c_minus_one` `\c_zero` `\c_sixteen` `\c_six` `\c_seven` `\c_twelve`

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 as TeX wants to reserve count registers 0–9, the first available one is 10 so we use that for `\c_minus_one`.

```
⟨*package⟩
\tex_let:D \c_minus_one \m@ne
⟨/package⟩
⟨*initex⟩
\tex_countdef:D \c_minus_one = 10 ~
\c_minus_one = -1 ~
⟨/initex⟩
\tex_chardef:D \c_sixteen = 16 ~
\tex_chardef:D \c_zero    = 0 ~
\tex_chardef:D \c_six     = 6 ~
\tex_chardef:D \c_seven   = 7 ~
\tex_chardef:D \c_twelve  = 12 ~
```

(*End definition for* `\c_minus_one`*,* `\c_zero`*, and* `\c_sixteen`*. These variables are documented on page 73.*)

`\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.

```
\etex_ifdefined:D \luatex_luatexversion:D
  \tex_chardef:D \c_max_register_int = 65 535 ~
\tex_else:D
  \tex_mathchardef:D \c_max_register_int = 32 767 ~
\tex_fi:D
```

(*End definition for* `\c_max_register_int`*. This variable is documented on page 73.*)

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.

```
\tex_let:D \cs_set_nopar:Npn           \tex_def:D
\tex_let:D \cs_set_nopar:Npx           \tex_edef:D
\etex_protected:D \cs_set_nopar:Npn \cs_set:Npn
```

```
  { \tex_long:D \cs_set_nopar:Npn }
\etex_protected:D \cs_set_nopar:Npn \cs_set:Npx
  { \tex_long:D \cs_set_nopar:Npx }
\etex_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npn
  { \etex_protected:D \cs_set_nopar:Npn }
\etex_protected:D \cs_set_nopar:Npn \cs_set_protected_nopar:Npx
  { \etex_protected:D \cs_set_nopar:Npx }
\cs_set_protected_nopar:Npn \cs_set_protected:Npn
  { \etex_protected:D \tex_long:D \cs_set_nopar:Npn }
\cs_set_protected_nopar:Npn \cs_set_protected:Npx
  { \etex_protected:D \tex_long:D \cs_set_nopar:Npx }
```

(*End definition for* `\cs_set_nopar:Npn` *and others. These functions are documented on page* **??**.)

\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.

```
\tex_let:D \cs_gset_nopar:Npn           \tex_gdef:D
\tex_let:D \cs_gset_nopar:Npx           \tex_xdef:D
\cs_set_protected_nopar:Npn \cs_gset:Npn
  { \tex_long:D \cs_gset_nopar:Npn }
\cs_set_protected_nopar:Npn \cs_gset:Npx
  { \tex_long:D \cs_gset_nopar:Npx }
\cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npn
  { \etex_protected:D \cs_gset_nopar:Npn }
\cs_set_protected_nopar:Npn \cs_gset_protected_nopar:Npx
  { \etex_protected:D \cs_gset_nopar:Npx }
\cs_set_protected_nopar:Npn \cs_gset_protected:Npn
  { \etex_protected:D \tex_long:D \cs_gset_nopar:Npn }
\cs_set_protected_nopar:Npn \cs_gset_protected:Npx
  { \etex_protected:D \tex_long:D \cs_gset_nopar:Npx }
```

(*End definition for* `\cs_gset_nopar:Npn` *and others. These functions are documented on page* **??**.)

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.

```
\cs_set_nopar:Npn \l__exp_internal_tl { }
```

(*End definition for* `\l__exp_internal_tl`*. This variable is documented on page 34.*)

\use:c

This macro grabs its argument and returns a csname from it.

```
\cs_set:Npn \use:c #1 { \cs:w #1 \cs_end: }
```

(*End definition for* `\use:c`*. This function is documented on page 17.*)

\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.

```
\cs_set_protected:Npn \use:x #1
  {
    \cs_set_nopar:Npx \l__exp_internal_tl {#1}
    \l__exp_internal_tl
  }
```

(*End definition for* `\use:x`*. This function is documented on page 20.*)

`\use:n`
`\use:nn`
`\use:nnn`
`\use:nnnn`

These macros grab their arguments and returns them back to the input (with outer braces removed).

```
\cs_set:Npn \use:n    #1       {#1}
\cs_set:Npn \use:nn   #1#2     {#1#2}
\cs_set:Npn \use:nnn  #1#2#3   {#1#2#3}
\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* ??.)

`\use_i:nn`
`\use_ii:nn`

The equivalent to LaTeX 2_ε's `\@firstoftwo` and `\@secondoftwo`.

```
\cs_set:Npn \use_i:nn  #1#2 {#1}
\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 19.*)

`\use_i:nnn`
`\use_ii:nnn`
`\use_iii:nnn`
`\use_i_ii:nnn`
`\use_i:nnnn`
`\use_ii:nnnn`
`\use_iii:nnnn`
`\use_iv:nnnn`

We also need something for picking up arguments from a longer list.

```
\cs_set:Npn \use_i:nnn    #1#2#3 {#1}
\cs_set:Npn \use_ii:nnn   #1#2#3 {#2}
\cs_set:Npn \use_iii:nnn  #1#2#3 {#3}
\cs_set:Npn \use_i_ii:nnn #1#2#3 {#1#2}
\cs_set:Npn \use_i:nnnn   #1#2#3#4 {#1}
\cs_set:Npn \use_ii:nnnn  #1#2#3#4 {#2}
\cs_set:Npn \use_iii:nnnn #1#2#3#4 {#3}
\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 19.*)

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

Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_recursion_stop`, respectively.

```
\cs_set:Npn \use_none_delimit_by_q_nil:w  #1 \q_nil  { }
\cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
\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_recurs`
These functions are documented on page 47.)

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

Same as above but execute first argument after gobbling. Very useful when you need to skip the rest of a mapping sequence but want an easy way to control what should be expanded next.

```
\cs_set:Npn \use_i_delimit_by_q_nil:nw  #1#2 \q_nil  {#1}
\cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
\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_sto`
These functions are documented on page 47.)

3.5 Gobbling tokens from input

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

To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of `n`'s following the `:` in the name. Although we could define functions to remove ten arguments or more using separate calls of `\use_none:nnnnn`, this is very non-intuitive to the programmer who will assume that expanding such a function once will take care of gobbling all the tokens in one go.

```
\cs_set:Npn \use_none:n         #1                 { }
\cs_set:Npn \use_none:nn        #1#2               { }
\cs_set:Npn \use_none:nnn       #1#2#3             { }
\cs_set:Npn \use_none:nnnn      #1#2#3#4           { }
\cs_set:Npn \use_none:nnnnn     #1#2#3#4#5         { }
\cs_set:Npn \use_none:nnnnnn    #1#2#3#4#5#6       { }
\cs_set:Npn \use_none:nnnnnnn   #1#2#3#4#5#6#7     { }
\cs_set:Npn \use_none:nnnnnnnn  #1#2#3#4#5#6#7#8   { }
\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* **??**.)

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 `\__int_to_roman:w` will expand fully any `\else:` and the `\fi:` that are waiting to be discarded, before reaching the `\c_zero` 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.

```
\cs_set_nopar:Npn \prg_return_true:
  { \exp_after:wN \use_i:nn  \__int_to_roman:w }
\cs_set_nopar:Npn \prg_return_false:
  { \exp_after:wN \use_ii:nn \__int_to_roman: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 37.*)

\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.

```
\cs_set_protected_nopar:Npn \prg_set_conditional:Npnn
  { \__prg_generate_conditional_parm:nnNpnn { set } { } }
\cs_set_protected_nopar:Npn \prg_new_conditional:Npnn
  { \__prg_generate_conditional_parm:nnNpnn { new } { } }
\cs_set_protected_nopar:Npn \prg_set_protected_conditional:Npnn
  { \__prg_generate_conditional_parm:nnNpnn { set } { _protected } }
\cs_set_protected_nopar:Npn \prg_new_protected_conditional:Npnn
  { \__prg_generate_conditional_parm:nnNpnn { new } { _protected } }
\cs_set_protected:Npn \__prg_generate_conditional_parm:nnNpnn #1#2#3#4#
  {
    \__cs_split_function:NN #3 \__prg_generate_conditional:nnNnnnnn
    {#1} {#2} {#4}
  }
```

(*End definition for* \prg_set_conditional:Npnn *and others. These functions are documented on page* *35.*)

\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.

```
\cs_set_protected_nopar:Npn \prg_set_conditional:Nnn
  { \__prg_generate_conditional_count:nnNnn { set } { } }
\cs_set_protected_nopar:Npn \prg_new_conditional:Nnn
  { \__prg_generate_conditional_count:nnNnn { new } { } }
\cs_set_protected_nopar:Npn \prg_set_protected_conditional:Nnn
  { \__prg_generate_conditional_count:nnNnn { set } { _protected } }
\cs_set_protected_nopar:Npn \prg_new_protected_conditional:Nnn
  { \__prg_generate_conditional_count:nnNnn { new } { _protected } }
\cs_set_protected:Npn \__prg_generate_conditional_count:nnNnn #1#2#3
  {
    \__cs_split_function:NN #3 \__prg_generate_conditional_count:nnNnnnn
```

```
    {#1} {#2}
  }
\cs_set_protected:Npn \__prg_generate_conditional_count:nnNnnnn #1#2#3#4#5
  {
    \__cs_parm_from_arg_count:nnF
      { \__prg_generate_conditional:nnNnnnnn {#1} {#2} #3 {#4} {#5} }
      { \tl_count:n {#2} }
      {
        \__msg_kernel_error:nnxx { kernel } { bad-number-of-arguments }
          { \token_to_str:c { #1 : #2 } }
          { \tl_count:n {#2} }
        \use_none:nn
      }
  }
```

(*End definition for* `\prg_set_conditional:Nnn` *and others. These functions are documented on page* ??.)

__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 `\etex_detokenize:D` makes the later loop more robust.

```
\cs_set_protected:Npn \__prg_generate_conditional:nnNnnnnn #1#2#3#4#5#6#7#8
  {
    \if_meaning:w \c_false_bool #3
      \__msg_kernel_error:nnx { kernel } { missing-colon }
        { \token_to_str:c {#1} }
      \exp_after:wN \use_none:nn
    \fi:
    \use:x
      {
        \exp_not:N \__prg_generate_conditional:nnnnnnw
        \exp_not:n { {#4} {#5} {#1} {#2} {#6} {#8} }
        \etex_detokenize:D {#7}
        \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
      }
  }
```

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.

```
\cs_set_protected:Npn \__prg_generate_conditional:nnnnnnw #1#2#3#4#5#6#7 ,
  {
    \if_meaning:w \q_recursion_tail #7
```

```
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
    \use:c { __prg_generate_ #7 _form:wnnnnnn }
        \tl_if_empty:nF {#7}
          {
            \__msg_kernel_error:nnxx
              { kernel } { conditional-form-unknown }
              {#7} { \token_to_str:c { #3 : #4 } }
          }
        \use_none:nnnnnnn
      \q_stop
      {#1} {#2} {#3} {#4} {#5} {#6}
    \__prg_generate_conditional:nnnnnnw {#1} {#2} {#3} {#4} {#5} {#6}
  }
```

(*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 `\c_zero`: 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.

```
\cs_set_protected:Npn \__prg_generate_p_form:wnnnnnn #1 \q_stop #2#3#4#5#6#7
  {
    \if_meaning:w \scan_stop: #3 \scan_stop:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
      {
        \exp_args:cc { cs_ #2 #3 :Npn } { #4 _p: #5 } #6
          { #7 \c_zero \c_true_bool \c_false_bool }
      }
      {
        \__msg_kernel_error:nnx { kernel } { protected-predicate }
          { \token_to_str:c { #4 _p: #5 } }
      }
  }
\cs_set_protected:Npn \__prg_generate_T_form:wnnnnnn #1 \q_stop #2#3#4#5#6#7
  {
    \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 T } #6
      { #7 \c_zero \use:n \use_none:n }
  }
\cs_set_protected:Npn \__prg_generate_F_form:wnnnnnn #1 \q_stop #2#3#4#5#6#7
  {
    \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 F } #6
      { #7 \c_zero { } }
  }
```

```
\cs_set_protected:Npn \__prg_generate_TF_form:wnnnnnn #1 \q_stop #2#3#4#5#6#7
  {
    \exp_args:cc { cs_ #2 #3 :Npn } { #4 : #5 TF } #6
      { #7 \c_zero }
  }
```

(*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 the two functions and feed a first auxiliary {⟨*name*$_1$⟩} {⟨*signature*$_1$⟩} ⟨*boolean*$_1$⟩ {⟨*name*$_2$⟩} {⟨*signature*$_2$⟩} ⟨*boolean*$_2$⟩ ⟨*copying function*⟩ ⟨*conditions*⟩ `, \q_recursion_tail , \q_recursion_stop`

```
\cs_set_protected_nopar:Npn \prg_set_eq_conditional:NNn
  { \__prg_set_eq_conditional:NNNn \cs_set_eq:cc }
\cs_set_protected_nopar:Npn \prg_new_eq_conditional:NNn
  { \__prg_set_eq_conditional:NNNn \cs_new_eq:cc }
\cs_set_protected:Npn \__prg_set_eq_conditional:NNNn #1#2#3#4
  {
    \use:x
      {
        \exp_not:N \__prg_set_eq_conditional:nnNnnNNw
          \__cs_split_function:NN #2 \prg_do_nothing:
          \__cs_split_function:NN #3 \prg_do_nothing:
          \exp_not:N #1
          \etex_detokenize:D {#4}
          \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
      }
  }
```

(*End definition for* `\prg_set_eq_conditional:NNn` *and* `\prg_new_eq_conditional:NNn`*. These functions are documented on page 37.*)

__prg_set_eq_conditional:nnNnnNNw
__prg_set_eq_conditional_loop:nnnnNw
__prg_set_eq_conditional_p_form:nnn
__prg_set_eq_conditional_TF_form:nnn
__prg_set_eq_conditional_T_form:nnn
__prg_set_eq_conditional_F_form:nnn

Split the function to be defined, and setup a manual clist loop over argument `#6` of the first auxiliary. The second auxiliary receives twice three arguments coming from splitting the function to be defined and the function to copy. Make sure that both functions contained a colon, otherwise we don't know how to build conditionals, hence abort. Call the looping macro, with arguments {⟨*name*$_1$⟩} {⟨*signature*$_1$⟩} {⟨*name*$_2$⟩} {⟨*signature*$_2$⟩} ⟨*copying function*⟩ and followed by the comma list. At each step in the loop, make sure that the conditional form we copy is defined, and copy it, otherwise abort.

```
\cs_set_protected:Npn \__prg_set_eq_conditional:nnNnnNNw #1#2#3#4#5#6
  {
    \if_meaning:w \c_false_bool #3
      \__msg_kernel_error:nnx { kernel } { missing-colon }
        { \token_to_str:c {#1} }
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
    \if_meaning:w \c_false_bool #6
      \__msg_kernel_error:nnx { kernel } { missing-colon }
        { \token_to_str:c {#4} }
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
    \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#4} {#5}
```

```
  }
\cs_set_protected:Npn \__prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 ,
  {
    \if_meaning:w \q_recursion_tail #6
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
    \use:c { __prg_set_eq_conditional_ #6 _form:wNnnnn }
        \tl_if_empty:nF {#6}
          {
            \__msg_kernel_error:nnxx
              { kernel } { conditional-form-unknown }
              {#6} { \token_to_str:c { #1 : #2 } }
          }
        \use_none:nnnnnn
      \q_stop
      #5 {#1} {#2} {#3} {#4}
    \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
  }
\cs_set:Npn \__prg_set_eq_conditional_p_form:wNnnnn #1 \q_stop #2#3#4#5#6
  {
    \__chk_if_exist_cs:c { #5 _p : #6    }
    #2 { #3 _p : #4    } { #5 _p : #6    }
  }
\cs_set:Npn \__prg_set_eq_conditional_TF_form:wNnnnn #1 \q_stop #2#3#4#5#6
  {
    \__chk_if_exist_cs:c { #5    : #6 TF }
    #2 { #3    : #4 TF } { #5    : #6 TF }
  }
\cs_set:Npn \__prg_set_eq_conditional_T_form:wNnnnn #1 \q_stop #2#3#4#5#6
  {
    \__chk_if_exist_cs:c { #5    : #6 T  }
    #2 { #3    : #4 T  } { #5    : #6 T  }
  }
\cs_set:Npn \__prg_set_eq_conditional_F_form:wNnnnn #1 \q_stop #2#3#4#5#6
  {
    \__chk_if_exist_cs:c { #5    : #6  F }
    #2 { #3    : #4  F } { #5    : #6  F }
  }
```

(*End definition for* `\__prg_set_eq_conditional:nnNnnNNw` *and* `\__prg_set_eq_conditional_loop:nnnnNw`.)

All that is left is to define the canonical boolean true and false. I think Michael originated the idea of expandable boolean tests. At first these were supposed to expand into either `TT` or `TF` to be tested using `\if:w` but this was later changed to `00` and `01`, so they could be used in logical operations. Later again they were changed to being numerical constants with values of 1 for true and 0 for false. We need this from the get-go.

`\c_true_bool`
`\c_false_bool`

Here are the canonical boolean values.

```
\tex_chardef:D \c_true_bool  = 1 ~
\tex_chardef:D \c_false_bool = 0 ~
```

(*End definition for* `\c_true_bool` *and* `\c_false_bool`. *These variables are documented on page 21.*)

3.7 Dissecting a control sequence

`\cs_to_str:N`
`\__cs_to_str:N`
`\__cs_to_str:w`

This converts a control sequence into the character string of its name, removing the leading escape character. This turns out to be a non-trivial matter as there a different cases:

- The usual case of a printable escape character;
- the case of a non-printable escape characters, e.g., when the value of the `\escapechar` is negative;
- when the escape character is a space.

One approach to solve this is to test how many tokens result from `\token_to_str:N \a`. If there are two tokens, then the escape character is printable, while if it is non-printable then only one is present.

However, there is an additional complication: the control sequence itself may start with a space. Clearly that should *not* be lost in the process of converting to a string. So the approach adopted is a little more intricate still. When the escape character is printable, `\token_to_str:N␣\␣` yields the escape character itself and a space. The character codes are different, thus the `\if:w` test is false, and TEX reads `\__cs_to_str:N` after turning the following control sequence into a string; this auxiliary removes the escape character, and stops the expansion of the initial `\__int_to_roman:w`. 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 `\__int_to_roman:w` 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 `\__int_to_roman:w` then sees `0`, which is not a terminated number, followed by the escape character, a space, which is removed, terminating the argument of `\__int_to_roman:w`. In all three cases, `\cs_to_str:N` takes two expansion steps to be fully expanded.

```
\cs_set_nopar:Npn \cs_to_str:N
  {
    \__int_to_roman:w
      \if:w \token_to_str:N \ \__cs_to_str:w \fi:
      \exp_after:wN \__cs_to_str:N \token_to_str:N
  }
\cs_set:Npn \__cs_to_str:N #1 { \c_zero }
\cs_set:Npn \__cs_to_str:w #1 \__cs_to_str:N
  { - \__int_value:w \fi: \exp_after:wN \c_zero }
```

(*End definition for* `\cs_to_str:N`. *This function is documented on page 18.*)

`\__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 can't use a literal `:` because it has the wrong catcode here, so it's transformed from `@` with `\tex_lowercase:D`.

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).

```
\group_begin:
\tex_lccode:D  `\@ = `\: \scan_stop:
\tex_catcode:D `\@ = 12 ~
\tex_lowercase:D
  {
    \group_end:
    \cs_set:Npn \__cs_split_function:NN #1
      {
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__cs_split_function_auxi:w
          \cs_to_str:N #1 \q_mark \c_true_bool
          @ \q_mark \c_false_bool
          \q_stop
      }
    \cs_set:Npn \__cs_split_function_auxi:w #1 @ #2 \q_mark #3#4 \q_stop #5
      { \__cs_split_function_auxii:w #5 #1 \q_mark \q_stop {#2} #3 }
    \cs_set:Npn \__cs_split_function_auxii:w #1#2 \q_mark #3 \q_stop
      { #1 {#2} }
  }
```

(*End definition for* `\__cs_split_function:NN`.)

`\__cs_get_function_name:N`
`\__cs_get_function_signature:N`

Simple wrappers.

```
\cs_set:Npn \__cs_get_function_name:N #1
  { \__cs_split_function:NN #1 \use_i:nnn }
\cs_set:Npn \__cs_get_function_signature:N #1
  { \__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
\cs_if_exist_p:c
\cs_if_exist:NTF
\cs_if_exist:cTF

Two versions for checking existence. For the N form we firstly check for \scan_stop: and then if it is in the hash table. There is no problem when inputting something like \else: or \fi: as TeX will only ever skip input in case the token tested against is \scan_stop:.

```
\prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \scan_stop:
      \prg_return_false:
    \else:
      \if_cs_exist:N #1
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    \fi:
  }
```

For the c form we firstly check if it is in the hash table and then for \scan_stop: so that we do not add it to the hash table unless it was already there. Here we have to be careful as the text to be skipped if the first test is false may contain tokens that disturb the scanner. Therefore, we ensure that the second test is performed after the first one has concluded completely.

```
\prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
  {
    \if_cs_exist:w #1 \cs_end:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
    {
      \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
        \prg_return_false:
      \else:
        \prg_return_true:
      \fi:
    }
    \prg_return_false:
  }
```

(*End definition for* \cs_if_exist:N *and* \cs_if_exist:c*. These functions are documented on page* ??.)

\cs_if_free_p:N
\cs_if_free_p:c
\cs_if_free:NTF
\cs_if_free:cTF

The logical reversal of the above.

```
\prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \scan_stop:
      \prg_return_true:
    \else:
      \if_cs_exist:N #1
        \prg_return_false:
      \else:
        \prg_return_true:
```

```
      \fi:
    \fi:
  }
\prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
  {
    \if_cs_exist:w #1 \cs_end:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
      {
        \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      }
      { \prg_return_true: }
  }
```

(*End definition for* `\cs_if_free:N` *and* `\cs_if_free:c`. *These functions are documented on page* **??**.)

`\cs_if_exist_use:NTF`
`\cs_if_exist_use:cTF`
`\cs_if_exist_use:N`
`\cs_if_exist_use:c`

The `\cs_if_exist_use:...` functions cannot be implemented as conditionals because the true branch must leave both the control sequence itself and the true code in the input stream. For the `c` variants, we are careful not to put the control sequence in the hash table if it does not exist.

```
\cs_set:Npn \cs_if_exist_use:NTF #1#2
  { \cs_if_exist:NTF #1 { #1 #2 } }
\cs_set:Npn \cs_if_exist_use:NF #1
  { \cs_if_exist:NTF #1 { #1 } }
\cs_set:Npn \cs_if_exist_use:NT #1 #2
  { \cs_if_exist:NTF #1 { #1 #2 } { } }
\cs_set:Npn \cs_if_exist_use:N #1
  { \cs_if_exist:NTF #1 { #1 } { } }
\cs_set:Npn \cs_if_exist_use:cTF #1#2
  { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } }
\cs_set:Npn \cs_if_exist_use:cF #1
  { \cs_if_exist:cTF {#1} { \use:c {#1} } }
\cs_set:Npn \cs_if_exist_use:cT #1#2
  { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } }
\cs_set:Npn \cs_if_exist_use:c #1
  { \cs_if_exist:cTF {#1} { \use:c {#1} } { } }
```

(*End definition for* `\cs_if_exist_use:NTF` *and* `\cs_if_exist_use:cTF`. *These functions are documented on page* **??**.)

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.

```
\cs_set_protected_nopar:Npn \iow_log:x
  { \tex_immediate:D \tex_write:D \c_minus_one }
\cs_set_protected_nopar:Npn \iow_term:x
  { \tex_immediate:D \tex_write:D \c_sixteen }
```

(*End definition for* `\iow_log:x` *and* `\iow_term:x`. *These functions are documented on page* **??**.)

`\__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.

```
\cs_set_protected:Npn \__msg_kernel_error:nnxx #1#2#3#4
  {
    \tex_errmessage:D
      {
        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^^J
        Argh,~internal~LaTeX3~error! ^^J ^^J
        Module ~ #1 , ~ message~name~"#2": ^^J
        Arguments~'#3'~and~'#4' ^^J ^^J
        This~is~one~for~The~LaTeX3~Project:~bailing~out
      }
    \tex_end:D
  }
\cs_set_protected:Npn \__msg_kernel_error:nnx #1#2#3
  { \__msg_kernel_error:nnxx {#1} {#2} {#3} { } }
\cs_set_protected:Npn \__msg_kernel_error:nn #1#2
  { \__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.

```
\cs_set_nopar:Npn \msg_line_context:
  { on~line~ \tex_the:D \tex_inputlineno:D }
```

(*End definition for* `\msg_line_context:`. *This function is documented on page 142.*)

`\__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!

```
\cs_set_protected:Npn \__chk_if_free_cs:N #1
  {
    \cs_if_free:NF #1
      {
```

```
        \__msg_kernel_error:nnxx { kernel } { command-already-defined }
          { \token_to_str:N #1 } { \token_to_meaning:N #1 }
      }
}
⟨*package⟩
\tex_ifodd:D \l@expl@log@functions@bool
  \cs_set_protected:Npn \__chk_if_free_cs:N #1
    {
      \cs_if_free:NF #1
        {
          \__msg_kernel_error:nnxx { kernel } { command-already-defined }
            { \token_to_str:N #1 } { \token_to_meaning:N #1 }
        }
      \iow_log:x { Defining~\token_to_str:N #1~ \msg_line_context: }
    }
\fi:
⟨/package⟩
\cs_set_protected_nopar:Npn \__chk_if_free_cs:c
  { \exp_args:Nc \__chk_if_free_cs:N }
```

(*End definition for* `\__chk_if_free_cs:N` *and* `\__chk_if_free_cs:c`.)

`\__chk_if_exist_var:N` Create the checking function for variable definitions when the option is set.

```
⟨*package⟩
\tex_ifodd:D \l@expl@check@declarations@bool
  \cs_set_protected:Npn \__chk_if_exist_var:N #1
    {
      \cs_if_exist:NF #1
        {
          \__msg_kernel_error:nnx { check } { non-declared-variable }
            { \token_to_str:N #1 }
        }
    }
\fi:
⟨/package⟩
```

(*End definition for* `\__chk_if_exist_var:N`.)

`\__chk_if_exist_cs:N`
`\__chk_if_exist_cs:c`
This function issues an error message when the control sequence in its argument does not exist.

```
\cs_set_protected:Npn \__chk_if_exist_cs:N #1
  {
    \cs_if_exist:NF #1
      {
        \__msg_kernel_error:nnx { kernel } { command-not-defined }
          { \token_to_str:N #1 }
      }
  }
\cs_set_protected_nopar:Npn \__chk_if_exist_cs:c
  { \exp_args:Nc \__chk_if_exist_cs:N }
```

(*End definition for* `\__chk_if_exist_cs:N` *and* `\__chk_if_exist_cs:c`.)

3.10 More new definitions

\cs_new_nopar:Npn
\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

Function which check that the control sequence is free before defining it.

```
\cs_set:Npn \__cs_tmp:w #1#2
  {
    \cs_set_protected:Npn #1 ##1
      {
        \__chk_if_free_cs:N ##1
        #2 ##1
      }
  }
\__cs_tmp:w \cs_new_nopar:Npn           \cs_gset_nopar:Npn
\__cs_tmp:w \cs_new_nopar:Npx           \cs_gset_nopar:Npx
\__cs_tmp:w \cs_new:Npn                 \cs_gset:Npn
\__cs_tmp:w \cs_new:Npx                 \cs_gset:Npx
\__cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
\__cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
\__cs_tmp:w \cs_new_protected:Npn       \cs_gset_protected:Npn
\__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* **??**.)

\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.

```
\cs_set:Npn \__cs_tmp:w #1#2
  { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
\__cs_tmp:w \cs_set_nopar:cpn  \cs_set_nopar:Npn
\__cs_tmp:w \cs_set_nopar:cpx  \cs_set_nopar:Npx
\__cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
\__cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
\__cs_tmp:w \cs_new_nopar:cpn  \cs_new_nopar:Npn
\__cs_tmp:w \cs_new_nopar:cpx  \cs_new_nopar:Npx
```

(*End definition for* `\cs_set_nopar:cpn` *and others. These functions are documented on page* **??**.)

\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.

```
\__cs_tmp:w \cs_set:cpn  \cs_set:Npn
\__cs_tmp:w \cs_set:cpx  \cs_set:Npx
\__cs_tmp:w \cs_gset:cpn \cs_gset:Npn
\__cs_tmp:w \cs_gset:cpx \cs_gset:Npx
\__cs_tmp:w \cs_new:cpn  \cs_new:Npn
\__cs_tmp:w \cs_new:cpx  \cs_new:Npx
```

(*End definition for* `\cs_set:cpn` *and others. These functions are documented on page* **??**.)

\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.

```
\__cs_tmp:w \cs_set_protected_nopar:cpn  \cs_set_protected_nopar:Npn
\__cs_tmp:w \cs_set_protected_nopar:cpx  \cs_set_protected_nopar:Npx
\__cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
\__cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
\__cs_tmp:w \cs_new_protected_nopar:cpn  \cs_new_protected_nopar:Npn
\__cs_tmp:w \cs_new_protected_nopar:cpx  \cs_new_protected_nopar:Npx
```

(*End definition for `\cs_set_protected_nopar:cpn` and others. These functions are documented on page* **??**.)

\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.

```
\__cs_tmp:w \cs_set_protected:cpn  \cs_set_protected:Npn
\__cs_tmp:w \cs_set_protected:cpx  \cs_set_protected:Npx
\__cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
\__cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
\__cs_tmp:w \cs_new_protected:cpn  \cs_new_protected:Npn
\__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* **??**.)

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".

```
\cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }
\cs_new_protected_nopar:Npn \cs_set_eq:cN { \exp_args:Nc  \cs_set_eq:NN }
\cs_new_protected_nopar:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
\cs_new_protected_nopar:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
\cs_new_protected_nopar:Npn \cs_gset_eq:NN { \tex_global:D  \cs_set_eq:NN }
\cs_new_protected_nopar:Npn \cs_gset_eq:Nc { \exp_args:NNc  \cs_gset_eq:NN }
\cs_new_protected_nopar:Npn \cs_gset_eq:cN { \exp_args:Nc   \cs_gset_eq:NN }
\cs_new_protected_nopar:Npn \cs_gset_eq:cc { \exp_args:Ncc  \cs_gset_eq:NN }
\cs_new_protected:Npn \cs_new_eq:NN #1
  {
    \__chk_if_free_cs:N #1
    \tex_global:D \cs_set_eq:NN #1
  }
\cs_new_protected_nopar:Npn \cs_new_eq:cN { \exp_args:Nc  \cs_new_eq:NN }
\cs_new_protected_nopar:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
\cs_new_protected_nopar:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }
```

(*End definition for `\cs_set_eq:NN` and others. These functions are documented on page* **??**.)

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.

```
\cs_new_protected:Npn \cs_undefine:N #1
  { \cs_gset_eq:NN #1 \tex_undefined:D }
\cs_new_protected:Npn \cs_undefine:c #1
  {
    \if_cs_exist:w #1 \cs_end:
      \exp_after:wN \use:n
    \else:
      \exp_after:wN \use_none:n
    \fi:
    { \cs_gset_eq:cN {#1} \tex_undefined:D }
  }
```

(*End definition for* \cs_undefine:N *and* \cs_undefine:c*. These functions are documented on page* ??.)

3.13 Generating parameter text from argument count

__cs_parm_from_arg_count:nnF
__cs_parm_from_arg_count_test:nnF

LaTeX3 provides shorthands to define control sequences and conditionals with a simple parameter text, derived directly from the signature, or more generally from knowing the number of arguments, between 0 and 9. This function expands to its first argument, untouched, followed by a brace group containing the parameter text {#1...#n}, where n is the result of evaluating the second argument (as described in \int_eval:n). If the second argument gives a result outside the range $[0,9]$, the third argument is returned instead, normally an error message. Some of the functions use here are not defined yet, but will be defined before this function is called.

```
\cs_set_protected:Npn \__cs_parm_from_arg_count:nnF #1#2
  {
    \exp_args:Nx \__cs_parm_from_arg_count_test:nnF
      {
        \exp_after:wN \exp_not:n
        \if_case:w \__int_eval:w #2 \__int_eval_end:
             { }
        \or: { ##1 }
        \or: { ##1##2 }
        \or: { ##1##2##3 }
        \or: { ##1##2##3##4 }
        \or: { ##1##2##3##4##5 }
        \or: { ##1##2##3##4##5##6 }
        \or: { ##1##2##3##4##5##6##7 }
        \or: { ##1##2##3##4##5##6##7##8 }
        \or: { ##1##2##3##4##5##6##7##8##9 }
        \else: { \c_false_bool }
        \fi:
      }
```

```
      {#1}
  }
\cs_set_protected:Npn \__cs_parm_from_arg_count_test:nnF #1#2
  {
    \if_meaning:w \c_false_bool #1
      \exp_after:wN \use_ii:nn
    \else:
      \exp_after:wN \use_i:nn
    \fi:
    { #2 {#1} }
  }
```

(*End definition for* `\__cs_parm_from_arg_count:nnF`.)

3.14 Defining functions from a given number of arguments

`\__cs_count_signature:N`
`\__cs_count_signature:c`
`\__cs_count_signature:nnN`

Counting the number of tokens in the signature, *i.e.*, the number of arguments the function should take. Since this is not used in any time-critical function, we simply use `\tl_count:n` if there is a signature, otherwise -1 arguments to signal an error. We need a variant form right away.

```
\cs_new:Npn \__cs_count_signature:N #1
  { \int_eval:n { \__cs_split_function:NN #1 \__cs_count_signature:nnN } }
\cs_new:Npn \__cs_count_signature:nnN #1#2#3
  {
    \if_meaning:w \c_true_bool #3
      \tl_count:n {#2}
    \else:
      \c_minus_one
    \fi:
  }
\cs_new_nopar:Npn \__cs_count_signature:c
  { \exp_args:Nc \__cs_count_signature:N }
```

(*End definition for* `\__cs_count_signature:N` *and* `\__cs_count_signature:c`.)

`\cs_generate_from_arg_count:NNnn`
`\cs_generate_from_arg_count:cNnn`
`\cs_generate_from_arg_count:Ncnn`

We provide a constructor function for defining functions with a given number of arguments. For this we need to choose the correct parameter text and then use that when defining. Since TeX supports from zero to nine arguments, we use a simple switch to choose the correct parameter text, ensuring the result is returned after finishing the conditional. If it is not between zero and nine, we throw an error.

1: function to define, 2: with what to define it, 3: the number of args it requires and 4: the replacement text

```
\cs_new_protected:Npn \cs_generate_from_arg_count:NNnn #1#2#3#4
  {
    \__cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
      {
        \__msg_kernel_error:nnxx { kernel } { bad-number-of-arguments }
          { \token_to_str:N #1 } { \int_eval:n {#3} }
      }
      {#4}
```

```
  }
```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```
\cs_new_protected_nopar:Npn \cs_generate_from_arg_count:cNnn
  { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
\cs_new_protected_nopar:Npn \cs_generate_from_arg_count:Ncnn
  { \exp_args:NNc \cs_generate_from_arg_count:NNnn }
```

(*End definition for* `\cs_generate_from_arg_count:NNnn`, `\cs_generate_from_arg_count:cNnn`, *and* `\cs_generate_from_arg_count:Ncnn`. *These functions are documented on page* ??.)

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. We define some simpler functions with user interface `\cs_set:Nn \foo_bar:nn {#1,#2}`, *i.e.*, the number of arguments is read from the signature.

\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
\cs_new_nopar:Nx
\cs_new_protected:Nn
\cs_new_protected:Nx
\cs_new_protected_nopar:Nn
\cs_new_protected_nopar:Nx

We want to define `\cs_set:Nn` as

```
\cs_set_protected:Npn \cs_set:Nn #1#2
  {
    \cs_generate_from_arg_count:NNnn #1 \cs_set:Npn
      { \__cs_count_signature:N #1 } {#2}
  }
```

In short, to define `\cs_set:Nn` we need just use `\cs_set:Npn`, everything else is the same for each variant. Therefore, we can make it simpler by temporarily defining a function to do this for us.

```
\cs_set:Npn \__cs_tmp:w #1#2#3
  {
    \cs_new_protected_nopar:cpx { cs_ #1 : #2 }
      {
        \exp_not:N \__cs_generate_from_signature:NNn
        \exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
      }
  }
\cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
  {
    \__cs_split_function:NN #2 \__cs_generate_from_signature:nnNNNn
    #1 #2
  }
\cs_new_protected:Npn \__cs_generate_from_signature:nnNNNn #1#2#3#4#5#6
  {
    \bool_if:NTF #3
      {
        \cs_generate_from_arg_count:NNnn
          #5 #4 { \tl_count:n {#2} } {#6}
      }
      {
```

```
        \__msg_kernel_error:nnx { kernel } { missing-colon }
          { \token_to_str:N #5 }
      }
  }
```

Then we define the 24 variants beginning with N.

```
\__cs_tmp:w { set }                   { Nn } { Npn }
\__cs_tmp:w { set }                   { Nx } { Npx }
\__cs_tmp:w { set_nopar }             { Nn } { Npn }
\__cs_tmp:w { set_nopar }             { Nx } { Npx }
\__cs_tmp:w { set_protected }         { Nn } { Npn }
\__cs_tmp:w { set_protected }         { Nx } { Npx }
\__cs_tmp:w { set_protected_nopar }   { Nn } { Npn }
\__cs_tmp:w { set_protected_nopar }   { Nx } { Npx }
\__cs_tmp:w { gset }                  { Nn } { Npn }
\__cs_tmp:w { gset }                  { Nx } { Npx }
\__cs_tmp:w { gset_nopar }            { Nn } { Npn }
\__cs_tmp:w { gset_nopar }            { Nx } { Npx }
\__cs_tmp:w { gset_protected }        { Nn } { Npn }
\__cs_tmp:w { gset_protected }        { Nx } { Npx }
\__cs_tmp:w { gset_protected_nopar }  { Nn } { Npn }
\__cs_tmp:w { gset_protected_nopar }  { Nx } { Npx }
\__cs_tmp:w { new }                   { Nn } { Npn }
\__cs_tmp:w { new }                   { Nx } { Npx }
\__cs_tmp:w { new_nopar }             { Nn } { Npn }
\__cs_tmp:w { new_nopar }             { Nx } { Npx }
\__cs_tmp:w { new_protected }         { Nn } { Npn }
\__cs_tmp:w { new_protected }         { Nx } { Npx }
\__cs_tmp:w { new_protected_nopar }   { Nn } { Npn }
\__cs_tmp:w { new_protected_nopar }   { Nx } { Npx }
```

(*End definition for* \cs_set:Nn *and others. These functions are documented on page* **??**.)

\cs_set:cn
\cs_set:cx
\cs_set_nopar:cn
\cs_set_nopar:cx
\cs_set_protected:cn
\cs_set_protected:cx
\cs_set_protected_nopar:cn
\cs_set_protected_nopar:cx
\cs_gset:cn
\cs_gset:cx
\cs_gset_nopar:cn
\cs_gset_nopar:cx
\cs_gset_protected:cn
\cs_gset_protected:cx
\cs_gset_protected_nopar:cn
\cs_gset_protected_nopar:cx
\cs_new:cn
\cs_new:cx
\cs_new_nopar:cn
\cs_new_nopar:cx
\cs_new_protected:cn
\cs_new_protected:cx
\cs_new_protected_nopar:cn
\cs_new_protected_nopar:cx

The 24 c variants simply use \exp_args:Nc.

```
\cs_set:Npn \__cs_tmp:w #1#2
  {
    \cs_new_protected_nopar:cpx { cs_ #1 : c #2 }
      {
        \exp_not:N \exp_args:Nc
        \exp_after:wN \exp_not:N \cs:w cs_ #1 : N #2 \cs_end:
      }
  }
\__cs_tmp:w { set }                   { n }
\__cs_tmp:w { set }                   { x }
\__cs_tmp:w { set_nopar }             { n }
\__cs_tmp:w { set_nopar }             { x }
\__cs_tmp:w { set_protected }         { n }
\__cs_tmp:w { set_protected }         { x }
\__cs_tmp:w { set_protected_nopar }   { n }
\__cs_tmp:w { set_protected_nopar }   { x }
\__cs_tmp:w { gset }                  { n }
```

```
\__cs_tmp:w { gset }                   { x }
\__cs_tmp:w { gset_nopar }             { n }
\__cs_tmp:w { gset_nopar }             { x }
\__cs_tmp:w { gset_protected }         { n }
\__cs_tmp:w { gset_protected }         { x }
\__cs_tmp:w { gset_protected_nopar } { n }
\__cs_tmp:w { gset_protected_nopar } { x }
\__cs_tmp:w { new }                    { n }
\__cs_tmp:w { new }                    { x }
\__cs_tmp:w { new_nopar }              { n }
\__cs_tmp:w { new_nopar }              { x }
\__cs_tmp:w { new_protected }          { n }
\__cs_tmp:w { new_protected }          { x }
\__cs_tmp:w { new_protected_nopar }  { n }
\__cs_tmp:w { new_protected_nopar }  { x }
```

(*End definition for* `\cs_set:cn` *and others. These functions are documented on page* **??**.)

3.16 Checking control sequence equality

`\cs_if_eq_p:NN`
`\cs_if_eq_p:cN`
`\cs_if_eq_p:Nc`
`\cs_if_eq_p:cc`
`\cs_if_eq:NNTF`
`\cs_if_eq:cNTF`
`\cs_if_eq:NcTF`
`\cs_if_eq:ccTF`

Check if two control sequences are identical.

```
\prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
  {
    \if_meaning:w #1#2
      \prg_return_true: \else: \prg_return_false: \fi:
  }
\cs_new_nopar:Npn \cs_if_eq_p:cN { \exp_args:Nc  \cs_if_eq_p:NN }
\cs_new_nopar:Npn \cs_if_eq:cNTF { \exp_args:Nc  \cs_if_eq:NNTF }
\cs_new_nopar:Npn \cs_if_eq:cNT  { \exp_args:Nc  \cs_if_eq:NNT }
\cs_new_nopar:Npn \cs_if_eq:cNF  { \exp_args:Nc  \cs_if_eq:NNF }
\cs_new_nopar:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
\cs_new_nopar:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
\cs_new_nopar:Npn \cs_if_eq:NcT  { \exp_args:NNc \cs_if_eq:NNT }
\cs_new_nopar:Npn \cs_if_eq:NcF  { \exp_args:NNc \cs_if_eq:NNF }
\cs_new_nopar:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
\cs_new_nopar:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
\cs_new_nopar:Npn \cs_if_eq:ccT  { \exp_args:Ncc \cs_if_eq:NNT }
\cs_new_nopar:Npn \cs_if_eq:ccF  { \exp_args:Ncc \cs_if_eq:NNF }
```

(*End definition for* `\cs_if_eq:NN` *and others. These functions are documented on page* **??**.)

3.17 Diagnostic functions

`\__kernel_register_show:N`
`\__kernel_register_show:c`

Check that the variable exists, then apply the `\showthe` primitive to the variable. The odd-looking `\use:n` gives a nicer output.

```
\cs_new_protected:Npn \__kernel_register_show:N #1
  {
    \cs_if_exist:NTF #1
      { \tex_showthe:D \use:n {#1} }
      {
```

```
      \__msg_kernel_error:nnx { kernel } { variable-not-defined }
        { \token_to_str:N #1 }
    }
  }
\cs_new_protected_nopar:Npn \__kernel_register_show:c
  { \exp_args:Nc \__kernel_register_show:N }
```

(*End definition for* `\__kernel_register_show:N` *and* `\__kernel_register_show:c`.)

`\cs_show:N`
`\cs_show:c`
`\__cs_show:www`

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: a line-break is added after the first colon in the meaning (this is what TeX does for macros and five `\...mark` primitives). 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.

```
\group_begin:
  \tex_lccode:D `? = `: \scan_stop:
  \tex_catcode:D `? = 12 \scan_stop:
\tex_lowercase:D
  {
    \group_end:
    \cs_new_protected:Npn \cs_show:N #1
      {
        \__msg_show_variable:n
          {
            > ~ \token_to_str:N #1 =
            \exp_after:wN \__cs_show:www \cs_meaning:N #1
              \use_none:nn ? \prg_do_nothing:
          }
      }
    \cs_new:Npn \__cs_show:www #1 ? { #1 ? \\ }
  }
\cs_new_protected_nopar:Npn \cs_show:c
  { \group_begin: \exp_args:NNc \group_end: \cs_show:N }
```

(*End definition for* `\cs_show:N` *and* `\cs_show:c`. *These functions are documented on page* **??**.)

3.18 Engine specific definitions

`\xetex_if_engine_p:`
`\luatex_if_engine_p:`
`\pdftex_if_engine_p:`
`\xetex_if_engine:TF`
`\luatex_if_engine:TF`
`\pdftex_if_engine:TF`

In some cases it will be useful to know which engine we're running. This can all be hard-coded for speed.

```
\cs_new_eq:NN \luatex_if_engine:T  \use_none:n
\cs_new_eq:NN \luatex_if_engine:F  \use:n
\cs_new_eq:NN \luatex_if_engine:TF \use_ii:nn
\cs_new_eq:NN \pdftex_if_engine:T  \use:n
\cs_new_eq:NN \pdftex_if_engine:F  \use_none:n
\cs_new_eq:NN \pdftex_if_engine:TF \use_i:nn
\cs_new_eq:NN \xetex_if_engine:T   \use_none:n
\cs_new_eq:NN \xetex_if_engine:F   \use:n
```

```
\cs_new_eq:NN \xetex_if_engine:TF  \use_ii:nn
\cs_new_eq:NN \luatex_if_engine_p: \c_false_bool
\cs_new_eq:NN \pdftex_if_engine_p: \c_true_bool
\cs_new_eq:NN \xetex_if_engine_p:  \c_false_bool
\cs_if_exist:NT \xetex_XeTeXversion:D
  {
    \cs_gset_eq:NN \pdftex_if_engine:T  \use_none:n
    \cs_gset_eq:NN \pdftex_if_engine:F  \use:n
    \cs_gset_eq:NN \pdftex_if_engine:TF \use_ii:nn
    \cs_gset_eq:NN \xetex_if_engine:T   \use:n
    \cs_gset_eq:NN \xetex_if_engine:F   \use_none:n
    \cs_gset_eq:NN \xetex_if_engine:TF  \use_i:nn
    \cs_gset_eq:NN \pdftex_if_engine_p: \c_false_bool
    \cs_gset_eq:NN \xetex_if_engine_p:  \c_true_bool
  }
\cs_if_exist:NT \luatex_directlua:D
  {
    \cs_gset_eq:NN \luatex_if_engine:T  \use:n
    \cs_gset_eq:NN \luatex_if_engine:F  \use_none:n
    \cs_gset_eq:NN \luatex_if_engine:TF \use_i:nn
    \cs_gset_eq:NN \pdftex_if_engine:T  \use_none:n
    \cs_gset_eq:NN \pdftex_if_engine:F  \use:n
    \cs_gset_eq:NN \pdftex_if_engine:TF \use_ii:nn
    \cs_gset_eq:NN \luatex_if_engine_p: \c_true_bool
    \cs_gset_eq:NN \pdftex_if_engine_p: \c_false_bool
  }
```

(*End definition for* `\xetex_if_engine:`*,* `\luatex_if_engine:`*, and* `\pdftex_if_engine:`*. These functions are documented on page 23.*)

3.19 Doing nothing functions

`\prg_do_nothing:`

This does not fit anywhere else!

```
\cs_new_nopar:Npn \prg_do_nothing: { }
```

(*End definition for* `\prg_do_nothing:`*. This function is documented on page 9.*)

3.20 String comparisons

`\str_if_eq_p:nn`
`\str_if_eq_x_p:nn`
`\str_if_eq:nnTF`
`\str_if_eq_x:nnTF`

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. These should eventually move somewhere else.

```
\prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
  {
    \if_int_compare:w \pdftex_strcmp:D { \exp_not:n {#1} } { \exp_not:n {#2} }
      = \c_zero
      \prg_return_true: \else: \prg_return_false: \fi:
  }
\prg_new_conditional:Npnn \str_if_eq_x:nn #1#2 { p , T , F , TF }
  {
```

```
    \if_int_compare:w \pdftex_strcmp:D {#1} {#2} = \c_zero
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\str_if_eq:nn` *and* `\str_if_eq_x:nn`*. These functions are documented on page* *22*.)

`\__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`, but hard-coded for speed.

```
\cs_new:Npn \__str_if_eq_x_return:nn #1 #2
  {
    \if_int_compare:w \pdftex_strcmp:D {#1} {#2} = \c_zero
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\__str_if_eq_x_return:nn`*.*)

`\str_case:nn`
`\str_case_x:nn`
`\str_case:nnTF`
`\str_case_x:nnTF`
`\__str_case:nnTF`
`\__str_case_x:nnTF`
`\__prg_case_end:nw`
`\__str_case:nw`
`\__str_case_x:nw`
`\__str_case_end:nw`

The aim here is to allow the case statement to be evaluated using a known number of expansion steps (two), and without needing to use an explicit "end of recursion" marker. That is achieved by using the test input as the final case, as this 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.

```
\cs_new:Npn \str_case:nn #1#2
  {
    \tex_romannumeral:D
    \__str_case:nnTF {#1} {#2} { } { }
  }
\cs_new:Npn \str_case:nnT #1#2#3
  {
    \tex_romannumeral:D
    \__str_case:nnTF {#1} {#2} {#3} { }
  }
\cs_new:Npn \str_case:nnF #1#2
  {
    \tex_romannumeral:D
    \__str_case:nnTF {#1} {#2} { }
  }
\cs_new:Npn \str_case:nnTF #1#2
  {
    \tex_romannumeral:D
    \__str_case:nnTF {#1} {#2}
  }
\cs_new:Npn \__str_case:nnTF #1#2#3#4
  { \__str_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
\cs_new:Npn \__str_case:nw #1#2#3
  {
    \str_if_eq:nnTF {#1} {#2}
      { \__str_case_end:nw {#3} }
```

```
      { \__str_case:nw {#1} }
  }
\cs_new:Npn \str_case_x:nn #1#2
  {
    \tex_romannumeral:D
    \__str_case_x:nnTF {#1} {#2} { } { }
  }
\cs_new:Npn \str_case_x:nnT #1#2#3
  {
    \tex_romannumeral:D
    \__str_case_x:nnTF {#1} {#2} {#3} { }
  }
\cs_new:Npn \str_case_x:nnF #1#2
  {
    \tex_romannumeral:D
    \__str_case_x:nnTF {#1} {#2} { }
  }
\cs_new:Npn \str_case_x:nnTF #1#2
  {
    \tex_romannumeral:D
    \__str_case_x:nnTF {#1} {#2}
  }
\cs_new:Npn \__str_case_x:nnTF #1#2#3#4
  { \__str_case_x:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
\cs_new:Npn \__str_case_x:nw #1#2#3
  {
    \str_if_eq_x:nnTF {#1} {#2}
      { \__str_case_end:nw {#3} }
      { \__str_case_x:nw {#1} }
  }
```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then `#1` 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`).

```
\cs_new:Npn \__prg_case_end:nw #1#2#3 \q_mark #4#5 \q_stop
  { \c_zero #1 #4 }
\cs_new_eq:NN \__str_case_end:nw \__prg_case_end:nw
```

(*End definition for* `\str_case:nn` *and* `\str_case_x:nn`*. These functions are documented on page* *26*.)

3.21 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.

```
\cs_new_eq:NN \__prg_break_point:Nn \use_ii:nn
\cs_new:Npn \__prg_map_break:Nn #1#2#3 \__prg_break_point:Nn #4#5
  {
    #5
    \if_meaning:w #1 #4
      \exp_after:wN \use_iii:nnn
    \fi:
    \__prg_map_break:Nn #1 {#2}
  }
```

(*End definition for* `\__prg_break_point:Nn` *and* `\__prg_map_break:Nn`*. These functions are documented on page 43.*)

`\__prg_break_point:`
`\__prg_break:`
`\__prg_break:n`

Very simple analogues of `\__prg_break_point:Nn` and `\__prg_map_break:Nn`, for use in fast short-term recursions which are not mappings, do not need to support nesting, and in which nothing has to be done at the end of the loop.

```
\cs_new_eq:NN \__prg_break_point: \prg_do_nothing:
\cs_new:Npn \__prg_break: #1 \__prg_break_point: { }
\cs_new:Npn \__prg_break:n #1#2 \__prg_break_point: {#1}
```

(*End definition for* `\__prg_break_point:`*. This function is documented on page* **??**.)

3.22 Deprecated functions

`\str_case:nnn`
`\str_case_x:nnn`

Deprecated 2013-07-15.

```
\cs_new_eq:NN \str_case:nnn \str_case:nnF
\cs_new_eq:NN \str_case_x:nnn \str_case_x:nnF
```

(*End definition for* `\str_case:nnn` *and* `\str_case_x:nnn`*. These functions are documented on page* **??**.)

```
⟨/initex | package⟩
```

4 l3expan implementation

```
⟨*initex | package⟩
⟨@@=exp⟩
```

`\exp_after:wN`
`\exp_not:N`
`\exp_not:n`

These are defined in l3basics.

(*End definition for* `\exp_after:wN`*. This function is documented on page 33.*)

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_nopar: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 variable is documented on page 34.*)

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.

```
\cs_new:Npn \__exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }
\cs_new:Npn \__exp_arg_next:Nnn #1#2#3 { #2 \::: { #3 #1 } }
```

(*End definition for* __exp_arg_next:nnn*.*)

\::: The end marker is just another name for the identity function.

```
\cs_new:Npn \::: #1 {#1}
```

(*End definition for* \:::*.*)

\::n This function is used to skip an argument that doesn't need to be expanded.

```
\cs_new:Npn \::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(*End 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.

```
\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.

```
\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.

```
\cs_new:Npn \::c #1 \::: #2#3
  { \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.

```
\cs_new:Npn \::o #1 \::: #2#3
  { \exp_after:wN \__exp_arg_next:nnn \exp_after:wN {#3} {#1} {#2} }
```

(*End definition for* \::o.)

\::f
\exp_stop_f: This function is used to expand a token list until the first unexpandable token is found. The underlying \romannumeral -`0 expands everything in its way to find something terminating the number and thereby expands the function in front of it. This scanning procedure is terminated once the expansion hits something non-expandable or a space. We introduce \exp_stop_f: to mark such an end of expansion marker; in case the scanner hits a number, this number also terminates the scanning and is left untouched. 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 \romannumeral -`0 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.

```
\cs_new:Npn \::f #1 \::: #2#3
  {
    \exp_after:wN \__exp_arg_next:nnn
      \exp_after:wN { \tex_romannumeral:D -`0 #3 }
      {#1} {#2}
  }
\use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }
```

(*End definition for* \::f.)

\::x This function is used to expand an argument fully.

```
\cs_new_protected:Npn \::x #1 \::: #2#3
  {
    \cs_set_nopar:Npx \l__exp_internal_tl { {#3} }
    \exp_after:wN \__exp_arg_next:nnn \l__exp_internal_tl {#1} {#2}
  }
```

(*End definition for* \::x.)

\::v
\::V These functions return the value of a register, i.e., one of tl, clist, int, skip, dim and 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 primitive \romannumeral sets off an expansion similar to an f type expansion, which we will terminate using \c_zero. The argument is returned in braces.

```
\cs_new:Npn \::V #1 \::: #2#3
  {
    \exp_after:wN \__exp_arg_next:nnn
      \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #3 }
      {#1} {#2}
}
\cs_new:Npn \::v # 1\::: #2#3
  {
```

```
    \exp_after:wN \__exp_arg_next:nnn
      \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:c {#3} }
      {#1} {#2}
  }
```

(*End definition for* `\::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:`.

```
\cs_new:Npn \__exp_eval_register:N #1
  {
    \exp_after:wN \if_meaning:w \exp_not:N #1 #1
```

If the token was not a macro it may be a malformed variable from a `c` expansion in which case it is equal to the primitive `\scan_stop:`. In that case we throw an error. We could let TeX do it for us but that would result in the rather obscure

```
! You can't use '\relax' after \the.
```

which while quite true doesn't give many hints as to what actually went wrong. We provide something more sensible.

```
      \if_meaning:w \scan_stop: #1
        \__exp_eval_error_msg:w
      \fi:
```

The next bit requires some explanation. The function must be initiated by the primitive `\romannumeral` and we want to terminate this expansion chain by inserting the `\c_zero` integer constant. 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 \c_zero \tex_the:D #1` and if it is a macro we need to execute `\exp_after:wN \c_zero #1`. We therefore issue the longer of the two sequences and if the register is a macro, we remove the `\tex_the:D`.

```
    \else:
      \exp_after:wN \use_i_ii:nnn
    \fi:
    \exp_after:wN \c_zero \tex_the:D #1
  }
\cs_new:Npn \__exp_eval_register:c #1
  { \exp_after:wN \__exp_eval_register:N \cs:w #1 \cs_end: }
```

Clean up nicely, then call the undefined control sequence. The result is an error message looking like this:

```
! Undefined control sequence.
<argument> \LaTeX3 error:
                            Erroneous variable used!
l.55 \tl_set:Nv \l_tmpa_tl {undefined_tl}
```

```
\cs_new:Npn \__exp_eval_error_msg:w #1 \tex_the:D #2
  {
      \fi:
    \fi:
    \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2}
    \c_zero
  }
```

(*End definition for* `\__exp_eval_register:N` *and* `\__exp_eval_register:c`.)

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!

```
\cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
\cs_new:Npn \exp_args:NNo #1#2#3
  { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
\cs_new:Npn \exp_args:NNNo #1#2#3#4
  { \exp_after:wN #1 \exp_after:wN#2 \exp_after:wN #3 \exp_after:wN {#4} }
```

(*End definition for* `\exp_args:No`. *This function is documented on page 31.*)

`\exp_args:Nc`
`\exp_args:cc`

In l3basics.

(*End definition for* `\exp_args:Nc` *and* `\exp_args:cc`. *These functions are documented on page* **??**.)

`\exp_args:NNc`
`\exp_args:Ncc`
`\exp_args:Nccc`

Here are the functions that turn their argument into csnames but are expandable.

```
\cs_new:Npn \exp_args:NNc #1#2#3
  { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
\cs_new:Npn \exp_args:Ncc #1#2#3
  { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
\cs_new:Npn \exp_args:Nccc #1#2#3#4
  {
    \exp_after:wN #1
      \cs:w #2 \exp_after:wN \cs_end:
      \cs:w #3 \exp_after:wN \cs_end:
      \cs:w #4 \cs_end:
  }
```

(*End definition for* `\exp_args:NNc`, `\exp_args:Ncc`, *and* `\exp_args:Nccc`. *These functions are documented on page* **??**.)

`\exp_args:Nf`
`\exp_args:NV`
`\exp_args:Nv`

```
\cs_new:Npn \exp_args:Nf #1#2
  { \exp_after:wN #1 \exp_after:wN { \tex_romannumeral:D -`0 #2 } }
\cs_new:Npn \exp_args:Nv #1#2
  {
    \exp_after:wN #1 \exp_after:wN
      { \tex_romannumeral:D \__exp_eval_register:c {#2} }
  }
\cs_new:Npn \exp_args:NV #1#2
```

```
  {
    \exp_after:wN #1 \exp_after:wN
      { \tex_romannumeral:D \__exp_eval_register:N #2 }
  }
```

(*End definition for* `\exp_args:Nf` *,* `\exp_args:NV` *, and* `\exp_args:Nv`*. These functions are documented on page* 30*.*)

`\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.

```
\cs_new:Npn \exp_args:NNf #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN { \tex_romannumeral:D -`0 #3 }
  }
\cs_new:Npn \exp_args:NNv #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:c {#3} }
  }
\cs_new:Npn \exp_args:NNV #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #3 }
  }
\cs_new:Npn \exp_args:Nco #1#2#3
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN {#3}
  }
\cs_new:Npn \exp_args:Ncf #1#2#3
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN { \tex_romannumeral:D -`0 #3 }
  }
\cs_new:Npn \exp_args:NVV #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN { \tex_romannumeral:D \exp_after:wN
      \__exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
    \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #3 }
  }
```

(*End definition for* `\exp_args:NNV` *and others. These functions are documented on page* ??*.*)

\exp_args:Ncco
\exp_args:NcNc
\exp_args:NcNo
\exp_args:NNNV

A few more that we can hand-tune.

```
\cs_new:Npn \exp_args:NNNV #1#2#3#4
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #4 }
  }
\cs_new:Npn \exp_args:NcNc #1#2#3#4
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN #3
    \cs:w #4 \cs_end:
  }
\cs_new:Npn \exp_args:NcNo #1#2#3#4
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \exp_after:wN #3
    \exp_after:wN {#4}
  }
\cs_new:Npn \exp_args:Ncco #1#2#3#4
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \cs:w #3 \exp_after:wN \cs_end:
    \exp_after:wN {#4}
  }
```

(*End definition for* `\exp_args:Ncco` *and others. These functions are documented on page* **??**.)

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

```
\cs_new_protected_nopar:Npn \exp_args:Nx { \::x \::: }
```

(*End definition for* `\exp_args:Nx`*. This function is documented on page 30.*)

\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.

```
\cs_new_nopar:Npn \exp_args:Nnc { \::n \::c \::: }
\cs_new_nopar:Npn \exp_args:Nfo { \::f \::o \::: }
\cs_new_nopar:Npn \exp_args:Nff { \::f \::f \::: }
\cs_new_nopar:Npn \exp_args:Nnf { \::n \::f \::: }
\cs_new_nopar:Npn \exp_args:Nno { \::n \::o \::: }
\cs_new_nopar:Npn \exp_args:NnV { \::n \::V \::: }
```

```
\cs_new_nopar:Npn \exp_args:Noo { \::o \::o \::: }
\cs_new_nopar:Npn \exp_args:Nof { \::o \::f \::: }
\cs_new_nopar:Npn \exp_args:Noc { \::o \::c \::: }
\cs_new_protected_nopar:Npn \exp_args:NNx { \::N \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Ncx { \::c \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Nnx { \::n \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Nox { \::o \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Nxo { \::x \::o \::: }
\cs_new_protected_nopar:Npn \exp_args:Nxx { \::x \::x \::: }
```

(*End definition for* `\exp_args:Nnc` *and others. These functions are documented on page* **??**.)

`\exp_args:NNno`
`\exp_args:NNoo`
`\exp_args:Nnnc`
`\exp_args:Nnno`
`\exp_args:Nooo`
`\exp_args:NNnx`
`\exp_args:NNox`
`\exp_args:Nnnx`
`\exp_args:Nnox`
`\exp_args:Nccx`
`\exp_args:Ncnx`
`\exp_args:Noox`

```
\cs_new_nopar:Npn \exp_args:NNno { \::N \::n \::o \::: }
\cs_new_nopar:Npn \exp_args:NNoo { \::N \::o \::o \::: }
\cs_new_nopar:Npn \exp_args:Nnnc { \::n \::n \::c \::: }
\cs_new_nopar:Npn \exp_args:Nnno { \::n \::n \::o \::: }
\cs_new_nopar:Npn \exp_args:Nooo { \::o \::o \::o \::: }
\cs_new_protected_nopar:Npn \exp_args:NNnx { \::N \::n \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:NNox { \::N \::o \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Nnnx { \::n \::n \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Nnox { \::n \::o \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Nccx { \::c \::c \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Ncnx { \::c \::n \::x \::: }
\cs_new_protected_nopar:Npn \exp_args:Noox { \::o \::o \::x \::: }
```

(*End definition for* `\exp_args:NNno` *and others. These functions are documented on page* **??**.)

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.

```
\cs_new:Npn \__exp_arg_last_unbraced:nn #1#2 { #2#1 }
\cs_new:Npn \::f_unbraced \::: #1#2
  {
    \exp_after:wN \__exp_arg_last_unbraced:nn
      \exp_after:wN { \tex_romannumeral:D -`0 #2 } {#1}
  }
\cs_new:Npn \::o_unbraced \::: #1#2
  { \exp_after:wN \__exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
\cs_new:Npn \::V_unbraced \::: #1#2
  {
    \exp_after:wN \__exp_arg_last_unbraced:nn
      \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:N #2 } {#1}
  }
\cs_new:Npn \::v_unbraced \::: #1#2
  {
    \exp_after:wN \__exp_arg_last_unbraced:nn
      \exp_after:wN { \tex_romannumeral:D \__exp_eval_register:c {#2} } {#1}
  }
\cs_new_protected:Npn \::x_unbraced \::: #1#2
```

```
  {
    \cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n {#1} #2 }
    \l__exp_internal_tl
  }
```

(*End definition for* `\__exp_arg_last_unbraced:nn`*.*)

`\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.

```
\cs_new:Npn \exp_last_unbraced:NV #1#2
  { \exp_after:wN #1 \tex_romannumeral:D \__exp_eval_register:N #2 }
\cs_new:Npn \exp_last_unbraced:Nv #1#2
  { \exp_after:wN #1 \tex_romannumeral:D \__exp_eval_register:c {#2} }
\cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
\cs_new:Npn \exp_last_unbraced:Nf #1#2
  { \exp_after:wN #1 \tex_romannumeral:D -`0 #2 }
\cs_new:Npn \exp_last_unbraced:Nco #1#2#3
  { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
\cs_new:Npn \exp_last_unbraced:NcV #1#2#3
  {
    \exp_after:wN #1
    \cs:w #2 \exp_after:wN \cs_end:
    \tex_romannumeral:D \__exp_eval_register:N #3
  }
\cs_new:Npn \exp_last_unbraced:NNV #1#2#3
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \tex_romannumeral:D \__exp_eval_register:N #3
  }
\cs_new:Npn \exp_last_unbraced:NNo #1#2#3
  { \exp_after:wN #1 \exp_after:wN #2 #3 }
\cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
  {
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \tex_romannumeral:D \__exp_eval_register:N #4
  }
\cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
  { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
\cs_new_nopar:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \::: }
\cs_new_nopar:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \::: }
\cs_new_nopar:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \::: }
\cs_new_nopar:Npn \exp_last_unbraced:NnNo { \::n \::N \::o_unbraced \::: }
\cs_new_protected_nopar:Npn \exp_last_unbraced:Nx { \::x_unbraced \::: }
```

(*End definition for* `\exp_last_unbraced:NV`*. This function is documented on page 32.*)

`\exp_last_two_unbraced:Noo`
`\__exp_last_two_unbraced:noN`

If `#2` is a single token then this can be implemented as

```
\cs_new:Npn \exp_last_two_unbraced:Noo #1 #2 #3
 { \exp_after:wN \exp_after:wN \exp_after:wN #1 \exp_after:wN #2 #3 }
```

However, for robustness this is not suitable. Instead, a bit of a shuffle is used to ensure that #2 can be multiple tokens.

```
\cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
  { \exp_after:wN \__exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
\cs_new:Npn \__exp_last_two_unbraced:noN #1#2#3
   { \exp_after:wN #3 #2 #1 }
```

(*End definition for* `\exp_last_two_unbraced:Noo`. *This function is documented on page 32.*)

4.5 Preventing expansion

\exp_not:o
\exp_not:c
\exp_not:f
\exp_not:V
\exp_not:v

```
\cs_new:Npn \exp_not:o #1 { \etex_unexpanded:D \exp_after:wN {#1} }
\cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
\cs_new:Npn \exp_not:f #1
  { \etex_unexpanded:D \exp_after:wN { \tex_romannumeral:D -`0 #1 } }
\cs_new:Npn \exp_not:V #1
  {
    \etex_unexpanded:D \exp_after:wN
      { \tex_romannumeral:D \__exp_eval_register:N #1 }
  }
\cs_new:Npn \exp_not:v #1
  {
    \etex_unexpanded:D \exp_after:wN
      { \tex_romannumeral:D \__exp_eval_register:c {#1} }
  }
```

(*End definition for* `\exp_not:o`. *This function is documented on page 33.*)

4.6 Defining function variants

```
⟨@@=cs⟩
```

\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_nopar:Npx` or `\cs_new_protected_nopar: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.

```
\cs_new_protected:Npn \cs_generate_variant:Nn #1#2
  {
    \__chk_if_exist_cs:N #1
    \__cs_generate_variant:N #1
    \exp_after:wN \__cs_split_function:NN
    \exp_after:wN #1
```

```
    \exp_after:wN \__cs_generate_variant:nnNN
    \exp_after:wN #1
    \etex_detokenize:D {#2} , \scan_stop: , \q_recursion_stop
  }
```

(*End definition for* `\cs_generate_variant:Nn`*. This function is documented on page 28.*)

`\__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_nopar:Npx`, otherwise it is `\cs_new_nopar:Npx`.

```
\group_begin:
  \tex_catcode:D `\M = 12 \scan_stop:
  \tex_catcode:D `\A = 12 \scan_stop:
  \tex_catcode:D `\P = 12 \scan_stop:
  \tex_catcode:D `\R = 12 \scan_stop:
\tex_lowercase:D
  {
    \group_end:
    \cs_new_protected:Npn \__cs_generate_variant:N #1
      {
        \exp_after:wN \if_meaning:w \exp_not:N #1 #1
          \cs_set_eq:NN \__cs_tmp:w \cs_new_protected_nopar:Npx
        \else:
          \exp_after:wN \__cs_generate_variant:ww
            \token_to_meaning:N #1 MA \q_mark
            \q_mark \cs_new_protected_nopar:Npx
            PR
            \q_mark \cs_new_nopar:Npx
            \q_stop
        \fi:
      }
    \cs_new_protected:Npn \__cs_generate_variant:ww #1 MA #2 \q_mark
      { \__cs_generate_variant:wwNw #1 }
    \cs_new_protected:Npn \__cs_generate_variant:wwNw
        #1 PR #2 \q_mark #3 #4 \q_stop
      {
        \cs_set_eq:NN \__cs_tmp:w #3
      }
```

```
  }
```

(*End definition for* `\__cs_generate_variant:N`.)

`\__cs_generate_variant:nnNN`

`#1` : Base name.
`#2` : Base signature.
`#3` : Boolean.
`#4` : Base function.

If the boolean is `\c_false_bool`, the base function has no colon and we abort with an error; otherwise, set off a loop through the desired variant forms. The original function is retained as `#4` for efficiency.

```
\cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
  {
    \if_meaning:w \c_false_bool #3
      \__msg_kernel_error:nnx { kernel } { missing-colon }
        { \token_to_str:c {#1} }
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
    \__cs_generate_variant:Nnnw #4 {#1}{#2}
  }
```

(*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.
- 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`.

```
\cs_new_protected:Npn \__cs_generate_variant:Nnnw #1#2#3#4 ,
  {
    \if_meaning:w \scan_stop: #4
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
    \use:x
      {
        \exp_not:N \__cs_generate_variant:wwNN
        \__cs_generate_variant_loop:nNwN { }
          #4
          \__cs_generate_variant_loop_end:nwwwNNnn
          \q_mark
          #3 ~
          { ~ { } \fi: \__cs_generate_variant_loop_long:wNNnn } ~
          { }
          \q_stop
        \exp_not:N #1 {#2} {#4}
      }
    \__cs_generate_variant:Nnnw #1 {#2} {#3}
  }
```

(*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.

```
\cs_new:Npn \__cs_generate_variant_loop:nNwN #1#2#3 \q_mark #4
  {
    \if:w #2 #4
      \exp_after:wN \__cs_generate_variant_loop_same:w
    \else:
      \if:w N #4 \else:
        \if:w n #4 \else:
          \__cs_generate_variant_loop_invalid:NNwNNnn #4#2
        \fi:
      \fi:
    \fi:
    #1
    \prg_do_nothing:
    #2
    \__cs_generate_variant_loop:nNwN { } #3 \q_mark
  }
\cs_new:Npn \__cs_generate_variant_loop_same:w
    #1 \prg_do_nothing: #2#3#4
  {
    #3 { #1 \__cs_generate_variant_same:N #2 }
  }
\cs_new:Npn \__cs_generate_variant_loop_end:nwwwNNnn
    #1#2 \q_mark #3 ~ #4 \q_stop #5#6#7#8
  {
    \scan_stop: \scan_stop: \fi:
    \exp_not:N \q_mark
    \exp_not:N \q_stop
    \exp_not:N #6
    \exp_not:c { #7 : #8 #1 #3 }
```

```
  }
\cs_new:Npn \__cs_generate_variant_loop_long:wNNnn #1 \q_stop #2#3#4#5
  {
    \exp_not:n
      {
        \q_mark
        \__msg_kernel_error:nnxx { kernel } { variant-too-long }
          {#5} { \token_to_str:N #3 }
        \use_none:nnnn
        \q_stop
        #3
        #3
      }
  }
\cs_new:Npn \__cs_generate_variant_loop_invalid:NNwNNnn
    #1#2 \fi: \fi: \fi: #3 \q_stop #4#5#6#7
  {
    \fi: \fi: \fi:
    \exp_not:n
      {
        \q_mark
        \__msg_kernel_error:nnxxxx { kernel } { invalid-variant }
          {#7} { \token_to_str:N #5 } {#1} {#2}
        \use_none:nnnn
        \q_stop
        #5
        #5
      }
  }
```

(*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.

```
\cs_new:Npn \__cs_generate_variant_same:N #1
  {
    \if:w N #1
      N
    \else:
      \if:w p #1
        p
      \else:
        n
      \fi:
    \fi:
  }
```

(*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_nopar:Npx. Then define the variant by combining the \exp_args:N #3 variant and the base function.

```
\cs_new_protected:Npn \__cs_generate_variant:wwNN
    #1 \q_mark #2 \q_stop #3#4
  {
    #2
    \cs_if_free:NTF #4
      {
        \group_begin:
          \__cs_generate_internal_variant:n {#1}
          \__cs_tmp:w #4 { \exp_not:c { exp_args:N #1 } \exp_not:N #3 }
        \group_end:
      }
      {
        \iow_log:x
          {
            Variant~\token_to_str:N #4~%
            already~defined;~ not~ changing~ it~on~line~%
            \tex_the:D \tex_inputlineno:D
          }
      }
  }
```

(*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.

```
\group_begin:
  \tex_catcode:D `\X = 12 \scan_stop:
  \tex_lccode:D `\N = `\N \scan_stop:
\tex_lowercase:D
  {
    \group_end:
    \cs_new_protected:Npn \__cs_generate_internal_variant:n #1
      {
        \__cs_generate_internal_variant:wwnNwnn
          #1 \q_mark
            { \cs_set_eq:NN \__cs_tmp:w \cs_new_protected_nopar:Npx }
            \cs_new_protected_nopar:cpx
          X \q_mark
            { }
            \cs_new_nopar:cpx
        \q_stop
          { exp_args:N #1 }
          { \__cs_generate_internal_variant_loop:n #1 { : \use_i:nn } }
      }
    \cs_new_protected:Npn \__cs_generate_internal_variant:wwnNwnn
        #1 X #2 \q_mark #3 #4 #5 \q_stop #6 #7
```

```
      {
        #3
        \cs_if_free:cT {#6} { #4 {#6} {#7} }
      }
  }
```

This command grabs char by char outputting `\::#1` (not expanded further). We avoid tests by putting a trailing `: \use_i:nn`, which leaves `\cs_end:` and removes the looping macro. The colon is in fact also turned into `\:::` so that the required structure for `\exp_args:N...` commands is correctly terminated.

```
\cs_new:Npn \__cs_generate_internal_variant_loop:n #1
  {
    \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
    \__cs_generate_internal_variant_loop:n
  }
```

(*End definition for* `\__cs_generate_internal_variant:n`*.*)

4.7 Variants which cannot be created earlier

`\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:VnTF`
`\str_if_eq:onTF`
`\str_if_eq:nVTF`
`\str_if_eq:noTF`
`\str_if_eq:VVTF`
`\str_case:on`
`\str_case:onTF`

These cannot come earlier as they need `\cs_generate_variant:Nn`.

```
\cs_generate_variant:Nn \str_if_eq_p:nn {  V ,  o }
\cs_generate_variant:Nn \str_if_eq_p:nn { nV , no , VV }
\cs_generate_variant:Nn \str_if_eq:nnT  {  V ,  o }
\cs_generate_variant:Nn \str_if_eq:nnT  { nV , no , VV }
\cs_generate_variant:Nn \str_if_eq:nnF  {  V ,  o }
\cs_generate_variant:Nn \str_if_eq:nnF  { nV , no , VV }
\cs_generate_variant:Nn \str_if_eq:nnTF {  V ,  o }
\cs_generate_variant:Nn \str_if_eq:nnTF { nV , no , VV }
\cs_generate_variant:Nn \str_case:nn   { o }
\cs_generate_variant:Nn \str_case:nnT  { o }
\cs_generate_variant:Nn \str_case:nnF  { o }
\cs_generate_variant:Nn \str_case:nnTF { o }
```

(*End definition for* `\str_if_eq:Vn` *and others. These functions are documented on page* **??***.*)

`\str_case:onn` Deprecated 2013-07-15.

```
\cs_new_eq:NN \str_case:onn \str_case:onF
```

(*End definition for* `\str_case:onn`*. This function is documented on page* **??***.*)

```
⟨/initex | package⟩
```

5 l3prg implementation

The following test files are used for this code: m3prg001.lvt,m3prg002.lvt,m3prg003.lvt.

```
⟨*initex | package⟩
```

5.1 Primitive conditionals

\if_bool:N
\if_predicate:w

Those two primitive TEX conditionals are synonyms. They should not be used outside the kernel code.

```
\tex_let:D \if_bool:N         \tex_ifodd:D
\tex_let:D \if_predicate:w    \tex_ifodd:D
```

(*End definition for* `\if_bool:N`. *This function is documented on page* *42*.)

5.2 Defining a set of conditional functions

\prg_set_conditional:Npnn
\prg_new_conditional:Npnn
\prg_set_protected_conditional:Npnn
\prg_new_protected_conditional:Npnn
\prg_set_conditional:Nnn
\prg_new_conditional:Nnn
\prg_set_protected_conditional:Nnn
\prg_new_protected_conditional:Nnn
\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn
\prg_return_true:
\prg_return_false:

These are all defined in l3basics, as they are needed "early". This is just a reminder!

(*End definition for* `\prg_set_conditional:Npnn` *and others. These functions are documented on page* *37*.)

5.3 The boolean data type

```
⟨@@=bool⟩
```

\bool_new:N
\bool_new:c

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

```
\cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
\cs_generate_variant:Nn \bool_new:N { c }
```

(*End definition for* `\bool_new:N` *and* `\bool_new:c`. *These functions are documented on page* **??**.)

\bool_set_true:N
\bool_set_true:c
\bool_gset_true:N
\bool_gset_true:c
\bool_set_false:N
\bool_set_false:c
\bool_gset_false:N
\bool_gset_false:c

Setting is already pretty easy.

```
\cs_new_protected:Npn \bool_set_true:N #1
  { \cs_set_eq:NN #1 \c_true_bool }
\cs_new_protected:Npn \bool_set_false:N #1
  { \cs_set_eq:NN #1 \c_false_bool }
\cs_new_protected:Npn \bool_gset_true:N #1
  { \cs_gset_eq:NN #1 \c_true_bool }
\cs_new_protected:Npn \bool_gset_false:N #1
  { \cs_gset_eq:NN #1 \c_false_bool }
\cs_generate_variant:Nn \bool_set_true:N   { c }
\cs_generate_variant:Nn \bool_set_false:N  { c }
\cs_generate_variant:Nn \bool_gset_true:N  { c }
\cs_generate_variant:Nn \bool_gset_false:N { c }
```

(*End definition for* `\bool_set_true:N` *and others. These functions are documented on page* **??**.)

\bool_set_eq:NN
\bool_set_eq:cN
\bool_set_eq:Nc
\bool_set_eq:cc
\bool_gset_eq:NN
\bool_gset_eq:cN
\bool_gset_eq:Nc
\bool_gset_eq:cc

The usual copy code.

```
\cs_new_eq:NN \bool_set_eq:NN  \cs_set_eq:NN
\cs_new_eq:NN \bool_set_eq:Nc  \cs_set_eq:Nc
\cs_new_eq:NN \bool_set_eq:cN  \cs_set_eq:cN
\cs_new_eq:NN \bool_set_eq:cc  \cs_set_eq:cc
\cs_new_eq:NN \bool_gset_eq:NN \cs_gset_eq:NN
\cs_new_eq:NN \bool_gset_eq:Nc \cs_gset_eq:Nc
\cs_new_eq:NN \bool_gset_eq:cN \cs_gset_eq:cN
\cs_new_eq:NN \bool_gset_eq:cc \cs_gset_eq:cc
```

(*End definition for* `\bool_set_eq:NN` *and others. These functions are documented on page* **??**.)

\bool_set:Nn
\bool_set:cn
\bool_gset:Nn
\bool_gset:cn

This function evaluates a boolean expression and assigns the first argument the meaning `\c_true_bool` or `\c_false_bool`.

```
\cs_new_protected:Npn \bool_set:Nn #1#2
  { \tex_chardef:D #1 = \bool_if_p:n {#2} }
\cs_new_protected:Npn \bool_gset:Nn #1#2
  { \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2} }
\cs_generate_variant:Nn \bool_set:Nn  { c }
\cs_generate_variant:Nn \bool_gset:Nn { c }
```

(*End definition for* `\bool_set:Nn` *and* `\bool_set:cn`*. These functions are documented on page* **??**.)

Booleans are not based on token lists but do need checking: this code complements similar material in l3tl.

```
⟨*package⟩
\tex_ifodd:D \l@expl@check@declarations@bool
  \cs_set_protected:Npn \bool_set_true:N #1
    {
      \__chk_if_exist_var:N #1
      \cs_set_eq:NN #1 \c_true_bool
    }
  \cs_set_protected:Npn \bool_set_false:N #1
    {
      \__chk_if_exist_var:N #1
      \cs_set_eq:NN #1 \c_false_bool
    }
  \cs_set_protected:Npn \bool_gset_true:N #1
    {
      \__chk_if_exist_var:N #1
      \cs_gset_eq:NN #1 \c_true_bool
    }
  \cs_set_protected:Npn \bool_gset_false:N #1
    {
      \__chk_if_exist_var:N #1
      \cs_gset_eq:NN #1 \c_false_bool
    }
  \cs_set_protected:Npn \bool_set_eq:NN  #1
    {
      \__chk_if_exist_var:N #1
      \cs_set_eq:NN #1
    }
  \cs_set_protected:Npn \bool_gset_eq:NN  #1
    {
      \__chk_if_exist_var:N #1
      \cs_gset_eq:NN #1
    }
  \cs_set_protected:Npn \bool_set:Nn #1#2
    {
      \__chk_if_exist_var:N #1
      \tex_chardef:D #1 = \bool_if_p:n {#2}
    }
  \cs_set_protected:Npn \bool_gset:Nn #1#2
```

```
    {
      \__chk_if_exist_var:N #1
      \tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}
    }
\tex_fi:D
⟨/package⟩
```

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

Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

```
\prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
  {
    \if_meaning:w \c_true_bool #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \bool_if_p:N { c }
\cs_generate_variant:Nn \bool_if:NT  { c }
\cs_generate_variant:Nn \bool_if:NF  { c }
\cs_generate_variant:Nn \bool_if:NTF { c }
```

(*End definition for* `\bool_if:N` *and* `\bool_if:c`*. These functions are documented on page* ??.)

\bool_show:N
\bool_show:c
\bool_show:n

Show the truth value of the boolean, as `true` or `false`. We use `\__msg_show_variable:n` to get a better output; this function requires its argument to start with `>~`.

```
\cs_new_protected:Npn \bool_show:N #1
  {
    \bool_if_exist:NTF #1
      { \bool_show:n {#1} }
      {
        \__msg_kernel_error:nnx { kernel } { variable-not-defined }
          { \token_to_str:N #1 }
      }
  }
\cs_new_protected:Npn \bool_show:n #1
  {
    \bool_if:nTF {#1}
      { \__msg_show_variable:n { > ~ true } }
      { \__msg_show_variable:n { > ~ false } }
  }
\cs_generate_variant:Nn \bool_show:N { c }
```

(*End definition for* `\bool_show:N`*,* `\bool_show:c`*, and* `\bool_show:n`*. These functions are documented on page* 38.)

\l_tmpa_bool
\l_tmpb_bool
\g_tmpa_bool
\g_tmpb_bool

A few booleans just if you need them.

```
\bool_new:N \l_tmpa_bool
\bool_new:N \l_tmpb_bool
\bool_new:N \g_tmpa_bool
\bool_new:N \g_tmpb_bool
```

(*End definition for* `\l_tmpa_bool` *and others. These variables are documented on page 38.*)

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

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \bool_if_exist:N \cs_if_exist:N { TF , T , F , p }
\prg_new_eq_conditional:NNn \bool_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(*End definition for* `\bool_if_exist:N` *and* `\bool_if_exist:c`*. These functions are documented on page* ??.)

5.4 Boolean expressions

`\bool_if_p:n`
`\bool_if:nTF`

Evaluating the truth value of a list of predicates is done using an input syntax somewhat similar to the one found in other programming languages with `(` and `)` for grouping, `!` for logical "Not", `&&` for logical "And" and `||` for logical "Or". 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*⟩.

⟨*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.

```
\prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
  {
    \if_predicate:w \bool_if_p:n {#1}
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\bool_if:n`*. These functions are documented on page 39.*)

`\bool_if_p:n`
`\__bool_if_left_parentheses:wwwn`
`\__bool_if_right_parentheses:wwwn`
`\__bool_if_or:wwwn`

First issue a `\group_align_safe_begin:` as we are using `&&` as syntax shorthand for the And operation and we need to hide it for TEX. This will be closed at the end of the expression parsing (see `S` below).

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.

```
\cs_new:Npn \bool_if_p:n #1
  {
    \group_align_safe_begin:
    \__bool_if_left_parentheses:wwwn \q_nil
      #1 \q_mark { }
      (  \q_mark { \__bool_if_right_parentheses:wwwn \q_nil }
      )  \q_mark { \__bool_if_or:wwwn \q_nil }
      || \q_mark \__bool_if_parse:NNNww
    \q_stop
  }
\cs_new:Npn \__bool_if_left_parentheses:wwwn #1 \q_nil #2 ( #3 \q_mark #4
  { #4 \__bool_if_left_parentheses:wwwn #1 #2 (( \q_nil #3 \q_mark {#4} }
\cs_new:Npn \__bool_if_right_parentheses:wwwn #1 \q_nil #2 ) #3 \q_mark #4
  { #4 \__bool_if_right_parentheses:wwwn #1 #2 )) \q_nil #3 \q_mark {#4} }
\cs_new:Npn \__bool_if_or:wwwn #1 \q_nil #2 || #3 \q_mark #4
  { #4 \__bool_if_or:wwwn #1 #2 )||( \q_nil #3 \q_mark {#4} }
```

(*End definition for* `\bool_if_p:n`*. This function is documented on page 39.*)

__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.

```
\cs_new:Npn \__bool_if_parse:NNNww #1#2#3#4 \q_mark #5 \q_stop
  {
    \__bool_get_next:NN \use_i:nn (( #4 )) S
  }
```

(*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.

```
\cs_new:Npn \__bool_get_next:NN #1#2
  {
    \use:c
      {
        __bool_
        \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
        :Nw
      }
      #1 #2
  }
```

(*End 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.

```
\cs_new:cpn { __bool_!:Nw } #1#2
  { \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.

```
\cs_new:cpn { __bool_(:Nw } #1#2
  {
    \exp_after:wN \__bool_choose:NNN \exp_after:wN #1
    \__int_value:w \__bool_get_next:NN \use_i:nn
  }
```

(*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.

```
\cs_new:cpn { __bool_p:Nw } #1
  { \exp_after:wN \__bool_choose:NNN \exp_after:wN #1 \__int_value:w }
```

(*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.

```
\cs_new:Npn \__bool_choose:NNN #1#2#3
  {
    \use:c
      {
        __bool_ #3 _
        #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: }
        :w
      }
  }
```

(*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.

```
\cs_new_nopar:cpn { __bool_)_0:w } { \c_false_bool }
\cs_new_nopar:cpn { __bool_)_1:w } { \c_true_bool }
\cs_new_nopar:cpn { __bool_S_0:w } { \group_align_safe_end: \c_false_bool }
\cs_new_nopar: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 `|`.

```
\cs_new_nopar:cpn { __bool_&_1:w } & { \__bool_get_next:NN \use_i:nn }
\cs_new_nopar:cpn { __bool_|_0:w } | { \__bool_get_next:NN \use_i:nn }
```

(*End definition for* `\__bool_&_1: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.

```
\cs_new_nopar:cpn { __bool_&_0:w } & { \__bool_eval_skip_to_end_auxi:Nw \c_false_bool }
\cs_new_nopar:cpn { __bool_|_1:w } | { \__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
```

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.

```
%% (
\cs_new:Npn \__bool_eval_skip_to_end_auxi:Nw #1#2 )
  {
    \__bool_eval_skip_to_end_auxii:Nw #1#2 ( % )
    \q_no_value \q_stop
    {#2}
  }
```

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

```
\cs_new:Npn \__bool_eval_skip_to_end_auxii:Nw #1#2 ( #3#4 \q_stop #5 % )
  {
    \quark_if_no_value:NTF #3
    {#1}
    { \__bool_eval_skip_to_end_auxiii:Nw #1 #5 }
  }
```

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!

```
\cs_new:Npn \__bool_eval_skip_to_end_auxiii:Nw #1#2 ( #3 )
  { % (
    \__bool_eval_skip_to_end_auxi:Nw #1#3 )
  }
```

(*End definition for* `\__bool_&_0:w`.)

\bool_not_p:n

The Not variant just reverses the outcome of `\bool_if_p:n`. Can be optimized but this is nice and simple and according to the implementation plan. Not even particularly useful to have it when the infix notation is easier to use.

```
\cs_new:Npn \bool_not_p:n #1 { \bool_if_p:n { ! ( #1 ) } }
```

(*End definition for* `\bool_not_p:n`*. This function is documented on page 40.*)

\bool_xor_p:nn

Exclusive or. If the boolean expressions have same truth value, return false, otherwise return true.

```
\cs_new:Npn \bool_xor_p:nn #1#2
  {
    \int_compare:nNnTF { \bool_if_p:n {#1} } = { \bool_if_p:n {#2} }
      \c_false_bool
      \c_true_bool
  }
```

(*End definition for* `\bool_xor_p:nn`*. This function is documented on page 40.*)

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.

```
\cs_new:Npn \bool_while_do:Nn #1#2
  { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
\cs_new:Npn \bool_until_do:Nn #1#2
  { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
\cs_generate_variant:Nn \bool_while_do:Nn { c }
\cs_generate_variant:Nn \bool_until_do:Nn { c }
```

(*End definition for* `\bool_while_do:Nn` *and* `\bool_while_do:cn`*. These functions are documented on page* **??**.)

\bool_do_while:Nn
\bool_do_while:cn
\bool_do_until:Nn
\bool_do_until:cn

A `do-while` loop where the body is performed at least once and the boolean is tested after executing the body. Otherwise identical to the above functions.

```
\cs_new:Npn \bool_do_while:Nn #1#2
  { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
\cs_new:Npn \bool_do_until:Nn #1#2
  { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
\cs_generate_variant:Nn \bool_do_while:Nn { c }
\cs_generate_variant:Nn \bool_do_until:Nn { c }
```

(*End definition for* `\bool_do_while:Nn` *and* `\bool_do_while:cn`*. These functions are documented on page* **??**.)

\bool_while_do:nn
\bool_do_while:nn
\bool_until_do:nn
\bool_do_until:nn

Loop functions with the test either before or after the first body expansion.

```
\cs_new:Npn \bool_while_do:nn #1#2
  {
    \bool_if:nT {#1}
      {
        #2
        \bool_while_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \bool_do_while:nn #1#2
  {
    #2
    \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
  }
\cs_new:Npn \bool_until_do:nn #1#2
  {
    \bool_if:nF {#1}
      {
        #2
        \bool_until_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \bool_do_until:nn #1#2
  {
    #2
    \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2}  }
  }
```

(*End definition for* `\bool_while_do:nn` *and others. These functions are documented on page 40.*)

5.6 Producing n copies

```
⟨@@=prg⟩
```

\prg_replicate:nn
__prg_replicate:N
__prg_replicate_first:N
__prg_replicate_
__prg_replicate_0:n
__prg_replicate_1:n
__prg_replicate_2:n
__prg_replicate_3:n
__prg_replicate_4:n
__prg_replicate_5:n
__prg_replicate_6:n
__prg_replicate_7:n
__prg_replicate_8:n
__prg_replicate_9:n
__prg_replicate_first_-:n
__prg_replicate_first_0:n
__prg_replicate_first_1:n
__prg_replicate_first_2:n
__prg_replicate_first_3:n
__prg_replicate_first_4:n
__prg_replicate_first_5:n
__prg_replicate_first_6:n
__prg_replicate_first_7:n
__prg_replicate_first_8:n
__prg_replicate_first_9:n

This function uses a cascading csname technique by David Kastrup (who else :-)

The idea is to make the input `25` result in first adding five, and then 20 copies of the code to be replicated. The technique uses cascading csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with `25` then the next function is the one that inserts two copies but it sees the ten copies handed down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. These functions also close the expansion of `\__int_to_roman: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 `\__int_to_roman:w` which can be done with just four macros but that method has its own problems since it can exhaust the string pool. Also, it is considerably slower than what we use here so the few extra csnames are well spent I would say.

```
\cs_new:Npn \prg_replicate:nn #1
  {
    \__int_to_roman:w
      \exp_after:wN \__prg_replicate_first:N
        \__int_value:w \__int_eval:w #1 \__int_eval_end:
      \cs_end:
  }
\cs_new:Npn \__prg_replicate:N #1
  { \cs:w __prg_replicate_#1 :n \__prg_replicate:N }
\cs_new:Npn \__prg_replicate_first:N #1
  { \cs:w __prg_replicate_first_ #1 :n \__prg_replicate:N }
```

Then comes all the functions that do the hard work of inserting all the copies. The first function takes `:n` as a parameter.

```
\cs_new:Npn \__prg_replicate_ :n #1 { \cs_end: }
\cs_new:cpn { __prg_replicate_0:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} }
\cs_new:cpn { __prg_replicate_1:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1 }
\cs_new:cpn { __prg_replicate_2:n } #1 { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1 }
\cs_new:cpn { __prg_replicate_3:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1 }
\cs_new:cpn { __prg_replicate_4:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1 }
\cs_new:cpn { __prg_replicate_5:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_6:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_7:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_8:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_9:n } #1
  { \cs_end: {#1#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1#1 }
```

Users shouldn't ask for something to be replicated once or even not at all but…

```
\cs_new:cpn { __prg_replicate_first_-:n } #1
  {
    \c_zero
    \__msg_kernel_expandable_error:nn { kernel } { negative-replication }
  }
\cs_new:cpn { __prg_replicate_first_0:n } #1 { \c_zero }
\cs_new:cpn { __prg_replicate_first_1:n } #1 { \c_zero #1 }
\cs_new:cpn { __prg_replicate_first_2:n } #1 { \c_zero #1#1 }
\cs_new:cpn { __prg_replicate_first_3:n } #1 { \c_zero #1#1#1 }
\cs_new:cpn { __prg_replicate_first_4:n } #1 { \c_zero #1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_5:n } #1 { \c_zero #1#1#1#1#1 }
```

```
\cs_new:cpn { __prg_replicate_first_6:n } #1 { \c_zero #1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_7:n } #1 { \c_zero #1#1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_8:n } #1 { \c_zero #1#1#1#1#1#1#1#1 }
\cs_new:cpn { __prg_replicate_first_9:n } #1 { \c_zero #1#1#1#1#1#1#1#1#1 }
```

(*End definition for* `\prg_replicate:nn`*. This function is documented on page 41.*)

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 `\c_zero` in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

```
\prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
  { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\mode_if_vertical:`*. These functions are documented on page 41.*)

\mode_if_horizontal_p:
\mode_if_horizontal:TF

For testing horizontal mode.

```
\prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
  { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\mode_if_horizontal:`*. These functions are documented on page 41.*)

\mode_if_inner_p:
\mode_if_inner:TF

For testing inner mode.

```
\prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
  { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\mode_if_inner:`*. These functions are documented on page 41.*)

\mode_if_math_p:
\mode_if_math:TF

For testing math mode. At the beginning of an alignment cell, the programmer should insert `\scan_align_safe_stop:` before the test.

```
\prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
  { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\mode_if_math:`*. These functions are documented on page 41.*)

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.

```
\cs_new_nopar:Npn \group_align_safe_begin:
  { \if_int_compare:w \if_false: { \fi: `} = \c_zero \fi: }
\cs_new_nopar:Npn \group_align_safe_end:
  { \if_int_compare:w `{ = \c_zero } \fi: }
```

(*End definition for* `\group_align_safe_begin:` *and* `\group_align_safe_end:`.)

`\scan_align_safe_stop:` When TeX is in the beginning of an align cell (right after the `\cr` or `&`) it is in a somewhat strange mode as it is looking ahead to find an `\omit` or `\noalign` and hasn't looked at the preamble yet. Thus an `\ifmmode` test at the start of an array cell (where math mode is introduced by the preamble, not in the cell itself) will always fail unless we stop TeX from scanning ahead. With ε-TeX's first version, this required inserting `\scan_stop:`, but not in all cases (see below). This is no longer needed with a newer ε-TeX, since protected macros are not expanded anymore at the beginning of an alignment cell. We can thus use an empty protected macro to stop TeX.

```
\cs_new_protected_nopar:Npn \scan_align_safe_stop: { }
```

Let us now explain the earlier version. We don't want to insert a `\scan_stop:` every time as that will destroy kerning between letters[3] Unfortunately there is no way to detect if we're in the beginning of an alignment cell as they have different characteristics depending on column number, *etc.* However we *can* detect if we're in an alignment cell by checking the current group type and we can also check if the previous node was a character or ligature. What is done here is that `\scan_stop:` is only inserted if an only if a) we're in the outer part of an alignment cell and b) the last node *wasn't* a char node or a ligature node. Thus an older definition here was

```
\cs_new_nopar:Npn \scan_align_safe_stop:
  {
    \int_compare:nNnT \etex_currentgrouptype:D = \c_six
      {
        \int_compare:nNnF \etex_lastnodetype:D = \c_zero
          {
            \int_compare:nNnF \etex_lastnodetype:D = \c_seven
              { \scan_stop: }
          }
      }
 }
```

However, this is not truly expandable, as there are places where the `\scan_stop:` ends up in the result.
(*End definition for* `\scan_align_safe_stop:`.)

```
⟨@@=prg⟩
```

[3] Unless we enforce an extra pass with an appropriate value of `\pretolerance`.

__prg_variable_get_scope:N
__prg_variable_get_scope:w
__prg_variable_get_type:N
__prg_variable_get_type:w

Expandable functions to find the type of a variable, and to return g if the variable is global. The trick for __prg_variable_get_scope:N is the same as that in __cs_split_function:NN, but it can be simplified as the requirements here are less complex.

```
\group_begin:
  \tex_lccode:D `* = `g \scan_stop:
  \tex_catcode:D `* = \c_twelve
\tl_to_lowercase:n
  {
    \group_end:
    \cs_new:Npn \__prg_variable_get_scope:N #1
      {
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__prg_variable_get_scope:w
          \cs_to_str:N #1 \exp_stop_f: \q_stop
      }
    \cs_new:Npn \__prg_variable_get_scope:w #1#2 \q_stop
      { \token_if_eq_meaning:NNT * #1 { g } }
  }
\group_begin:
  \tex_lccode:D `* = `_ \scan_stop:
  \tex_catcode:D `* = \c_twelve
\tl_to_lowercase:n
  {
    \group_end:
    \cs_new:Npn \__prg_variable_get_type:N #1
      {
        \exp_after:wN \__prg_variable_get_type:w
          \token_to_str:N #1 * a \q_stop
      }
    \cs_new:Npn \__prg_variable_get_type:w #1 * #2#3 \q_stop
      {
        \token_if_eq_meaning:NNTF a #2
          {#1}
          { \__prg_variable_get_type:w #2#3 \q_stop }
      }
  }
```

(*End definition for __prg_variable_get_scope:N.*)

\g__prg_map_int

A nesting counter for mapping.

```
\int_new:N \g__prg_map_int
```

(*End definition for \g__prg_map_int. This variable is documented on page 43.*)

__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. This function is documented on page 43.*)

__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:. This function is documented on page ??.*)

```
⟨/initex | package⟩
```

6 l3quark implementation

The following test files are used for this code: m3quark001.lvt.

```
⟨*initex | package⟩
```

6.1 Quarks

\quark_new:N Allocate a new quark.

```
\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 45.*)

\q_nil
\q_mark
\q_no_value
\q_stop

Some "public" quarks. `\q_stop` is an "end of argument" marker, `\q_nil` is a empty value and `\q_no_value` marks an empty argument.

```
\quark_new:N \q_nil
\quark_new:N \q_mark
\quark_new:N \q_no_value
\quark_new:N \q_stop
```

(*End definition for* `\q_nil` *and others. These variables are documented on page 45.*)

\q_recursion_tail
\q_recursion_stop

Quarks for ending recursions. Only ever used there! `\q_recursion_tail` is appended to whatever list structure we are doing recursion on, meaning it is added as a proper list item with whatever list separator is in use. `\q_recursion_stop` is placed directly after the list.

```
\quark_new:N \q_recursion_tail
\quark_new:N \q_recursion_stop
```

(*End definition for* `\q_recursion_tail` *and* `\q_recursion_stop`*. These variables are documented on page 46.*)

\quark_if_recursion_tail_stop:N
\quark_if_recursion_tail_stop_do:Nn

When doing recursions, it is easy to spend a lot of time testing if the end marker has been found. To avoid this, a dedicated end marker is used each time a recursion is set up. Thus if the marker is found everything can be wrapper up and finished off. The simple case is when the test can guarantee that only a single token is being tested. In this case, there is just a dedicated copy of the standard quark test. Both a gobbling version and one inserting end code are provided.

```
\cs_new:Npn \quark_if_recursion_tail_stop:N #1
  {
    \if_meaning:w \q_recursion_tail #1
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
  }
\cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
  {
    \if_meaning:w \q_recursion_tail #1
      \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
    \else:
      \exp_after:wN \use_none:n
    \fi:
  }
```

(*End definition for* `\quark_if_recursion_tail_stop:N`. *This function is documented on page 46.*)

\quark_if_recursion_tail_stop:n
\quark_if_recursion_tail_stop:o
\quark_if_recursion_tail_stop_do:nn
\quark_if_recursion_tail_stop_do:on

The same idea applies when testing multiple tokens, but here we just compare the token list to `\q_recursion_tail` as a string.

```
\cs_new:Npn \quark_if_recursion_tail_stop:n #1
  {
    \if_int_compare:w \pdftex_strcmp:D
      { \exp_not:N \q_recursion_tail } { \exp_not:n {#1} } = \c_zero
      \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
    \fi:
  }
\cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
  {
    \if_int_compare:w \pdftex_strcmp:D
      { \exp_not:N \q_recursion_tail } { \exp_not:n {#1} } = \c_zero
      \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
    \else:
      \exp_after:wN \use_none:n
    \fi:
  }
\cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
\cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn { o }
```

(*End definition for* `\quark_if_recursion_tail_stop:n` *and* `\quark_if_recursion_tail_stop:o`. *These functions are documented on page* **??**.)

__quark_if_recursion_tail_break:NN
__quark_if_recursion_tail_break:nN

Analogs of the `\quark_if_recursion_tail_stop...` functions. Break the mapping using `#2`.

```
\cs_new:Npn \__quark_if_recursion_tail_break:NN #1#2
  {
    \if_meaning:w \q_recursion_tail #1
      \exp_after:wN #2
    \fi:
  }
\cs_new:Npn \__quark_if_recursion_tail_break:nN #1#2
  {
    \if_int_compare:w \pdftex_strcmp:D
      { \exp_not:N \q_recursion_tail } { \exp_not:n {#1} } = \c_zero
      \exp_after:wN #2
    \fi:
  }
```

(*End definition for* `\__quark_if_recursion_tail_break:NN`. *This function is documented on page* **??**.)

\quark_if_nil_p:N
\quark_if_nil:N*TF*
\quark_if_no_value_p:N
\quark_if_no_value_p:c
\quark_if_no_value:N*TF*
\quark_if_no_value:c*TF*

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.[4]

```
\prg_new_conditional:Nnn \quark_if_nil:N { p, T , F , TF }
  {
```

[4]It may still loop in special circumstances however!

```
    \if_meaning:w \q_nil #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_new_conditional:Nnn \quark_if_no_value:N { p, T , F , TF }
  {
    \if_meaning:w \q_no_value #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \quark_if_no_value_p:N { c }
\cs_generate_variant:Nn \quark_if_no_value:NT  { c }
\cs_generate_variant:Nn \quark_if_no_value:NF  { c }
\cs_generate_variant:Nn \quark_if_no_value:NTF { c }
```

(*End definition for* `\quark_if_nil:N`. *These functions are documented on page* **??**.)

`\quark_if_nil_p:n`
`\quark_if_nil_p:V`
`\quark_if_nil_p:o`
`\quark_if_nil:nTF`
`\quark_if_nil:VTF`
`\quark_if_nil:oTF`
`\quark_if_no_value_p:n`
`\quark_if_no_value:nTF`

These are essentially `\str_if_eq:nn` tests but done directly.

```
\prg_new_conditional:Nnn \quark_if_nil:n { p, T , F , TF }
  {
    \if_int_compare:w \pdftex_strcmp:D
      { \exp_not:N \q_nil } { \exp_not:n {#1} } = \c_zero
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_new_conditional:Nnn \quark_if_no_value:n { p, T , F , TF }
  {
    \if_int_compare:w \pdftex_strcmp:D
      { \exp_not:N \q_no_value } { \exp_not:n {#1} } = \c_zero
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \quark_if_nil_p:n { V , o }
\cs_generate_variant:Nn \quark_if_nil:nTF { V , o }
\cs_generate_variant:Nn \quark_if_nil:nT  { V , o }
\cs_generate_variant:Nn \quark_if_nil:nF  { V , o }
```

(*End definition for* `\quark_if_nil:n`, `\quark_if_nil:V`, *and* `\quark_if_nil:o`. *These functions are documented on page 45.*)

`\q__tl_act_mark`
`\q__tl_act_stop`

These private quarks are needed by l3tl, but that is loaded before the quark module, hence their definition is deferred.

```
\quark_new:N \q__tl_act_mark
\quark_new:N \q__tl_act_stop
```

(*End definition for* `\q__tl_act_mark` *and* `\q__tl_act_stop`*. These variables are documented on page* ??*.*)

6.2 Scan marks

```
⟨@@=scan⟩
```

`\g__scan_marks_tl` The list of all scan marks currently declared.

```
\tl_new:N \g__scan_marks_tl
```

(*End definition for* `\g__scan_marks_tl`*. This variable is documented on page* ??*.*)

`\__scan_new:N` Check whether the variable is already a scan mark, then declare it to be equal to `\scan_stop:` globally.

```
\cs_new_protected:Npn \__scan_new:N #1
  {
    \tl_if_in:NnTF \g__scan_marks_tl { #1 }
      {
        \__msg_kernel_error:nnx { kernel } { scanmark-already-defined }
          { \token_to_str:N #1 }
      }
      {
        \tl_gput_right:Nn \g__scan_marks_tl {#1}
        \cs_new_eq:NN #1 \scan_stop:
      }
  }
```

(*End definition for* `\__scan_new:N`*.*)

`\s__stop` We only declare one scan mark here, more can be defined by specific modules.

```
\__scan_new:N \s__stop
```

(*End definition for* `\s__stop`*. This variable is documented on page 48.*)

`\__use_none_delimit_by_s__stop:w` Similar to `\use_none_delimit_by_q_stop:w`.

```
\cs_new:Npn \__use_none_delimit_by_s__stop:w #1 \s__stop { }
```

(*End 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.

```
\__scan_new:N \s__seq
```

(*End definition for* `\s__seq`*. This variable is documented on page 112.*)

6.3 Deprecated quark functions

\quark_if_recursion_tail_break:N
\quark_if_recursion_tail_break:n

It's not clear what breaking function we should be using here, so I'm picking one somewhat arbitrarily.

```
\cs_new:Npn \quark_if_recursion_tail_break:N #1
  { \__quark_if_recursion_tail_break:NN #1 \prg_break: }
\cs_new:Npn \quark_if_recursion_tail_break:n #1
  { \__quark_if_recursion_tail_break:nN {#1} \prg_break: }
```

(*End definition for* `\quark_if_recursion_tail_break:N` *and* `\quark_if_recursion_tail_break:n`*. These functions are documented on page* ??.)

```
⟨/initex | package⟩
```

7 l3token implementation

```
⟨*initex | package⟩
⟨@@=token⟩
```

7.1 Character tokens

\char_set_catcode:nn
\char_value_catcode:n
\char_show_value_catcode:n

Category code changes.

```
\cs_new_protected:Npn \char_set_catcode:nn #1#2
  { \tex_catcode:D #1 = \__int_eval:w #2 \__int_eval_end: }
\cs_new:Npn \char_value_catcode:n #1
  { \tex_the:D \tex_catcode:D \__int_eval:w #1\__int_eval_end: }
\cs_new_protected:Npn \char_show_value_catcode:n #1
  { \tex_showthe:D \tex_catcode:D \__int_eval:w #1 \__int_eval_end: }
```

(*End definition for* `\char_set_catcode:nn`*. This function is documented on page 51.*)

\char_set_catcode_escape:N
\char_set_catcode_group_begin:N
\char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N
\char_set_catcode_alignment:N
\char_set_catcode_end_line:N
\char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
\char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N

```
\cs_new_protected:Npn \char_set_catcode_escape:N #1
  { \char_set_catcode:nn { `#1 } \c_zero }
\cs_new_protected:Npn \char_set_catcode_group_begin:N #1
  { \char_set_catcode:nn { `#1 } \c_one }
\cs_new_protected:Npn \char_set_catcode_group_end:N #1
  { \char_set_catcode:nn { `#1 } \c_two }
\cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
  { \char_set_catcode:nn { `#1 } \c_three }
\cs_new_protected:Npn \char_set_catcode_alignment:N #1
  { \char_set_catcode:nn { `#1 } \c_four }
\cs_new_protected:Npn \char_set_catcode_end_line:N #1
  { \char_set_catcode:nn { `#1 } \c_five }
\cs_new_protected:Npn \char_set_catcode_parameter:N #1
  { \char_set_catcode:nn { `#1 } \c_six }
\cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
  { \char_set_catcode:nn { `#1 } \c_seven }
\cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
  { \char_set_catcode:nn { `#1 } \c_eight }
\cs_new_protected:Npn \char_set_catcode_ignore:N #1
```

```
  { \char_set_catcode:nn { `#1 } \c_nine }
\cs_new_protected:Npn \char_set_catcode_space:N #1
  { \char_set_catcode:nn { `#1 } \c_ten }
\cs_new_protected:Npn \char_set_catcode_letter:N #1
  { \char_set_catcode:nn { `#1 } \c_eleven }
\cs_new_protected:Npn \char_set_catcode_other:N #1
  { \char_set_catcode:nn { `#1 } \c_twelve }
\cs_new_protected:Npn \char_set_catcode_active:N #1
  { \char_set_catcode:nn { `#1 } \c_thirteen }
\cs_new_protected:Npn \char_set_catcode_comment:N #1
  { \char_set_catcode:nn { `#1 } \c_fourteen }
\cs_new_protected:Npn \char_set_catcode_invalid:N #1
  { \char_set_catcode:nn { `#1 } \c_fifteen }
```

(*End definition for* `\char_set_catcode_escape:N` *and others. 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

```
\cs_new_protected:Npn \char_set_catcode_escape:n #1
  { \char_set_catcode:nn {#1} \c_zero }
\cs_new_protected:Npn \char_set_catcode_group_begin:n #1
  { \char_set_catcode:nn {#1} \c_one }
\cs_new_protected:Npn \char_set_catcode_group_end:n #1
  { \char_set_catcode:nn {#1} \c_two }
\cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
  { \char_set_catcode:nn {#1} \c_three }
\cs_new_protected:Npn \char_set_catcode_alignment:n #1
  { \char_set_catcode:nn {#1} \c_four }
\cs_new_protected:Npn \char_set_catcode_end_line:n #1
  { \char_set_catcode:nn {#1} \c_five }
\cs_new_protected:Npn \char_set_catcode_parameter:n #1
  { \char_set_catcode:nn {#1} \c_six }
\cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
  { \char_set_catcode:nn {#1} \c_seven }
\cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
  { \char_set_catcode:nn {#1} \c_eight }
\cs_new_protected:Npn \char_set_catcode_ignore:n #1
  { \char_set_catcode:nn {#1} \c_nine }
\cs_new_protected:Npn \char_set_catcode_space:n #1
  { \char_set_catcode:nn {#1} \c_ten }
\cs_new_protected:Npn \char_set_catcode_letter:n #1
  { \char_set_catcode:nn {#1} \c_eleven }
\cs_new_protected:Npn \char_set_catcode_other:n #1
  { \char_set_catcode:nn {#1} \c_twelve }
\cs_new_protected:Npn \char_set_catcode_active:n #1
  { \char_set_catcode:nn {#1} \c_thirteen }
\cs_new_protected:Npn \char_set_catcode_comment:n #1
  { \char_set_catcode:nn {#1} \c_fourteen }
\cs_new_protected:Npn \char_set_catcode_invalid:n #1
  { \char_set_catcode:nn {#1} \c_fifteen }
```

(*End definition for* `\char_set_catcode_escape:n` *and others. These functions are documented on page 50.*)

\char_set_mathcode:nn
\char_value_mathcode:n
\char_show_value_mathcode:n
\char_set_lccode:nn
\char_value_lccode:n
\char_show_value_lccode:n
\char_set_uccode:nn
\char_value_uccode:n
\char_show_value_uccode:n
\char_set_sfcode:nn
\char_value_sfcode:n
\char_show_value_sfcode:n

Pretty repetitive, but necessary!

```
\cs_new_protected:Npn \char_set_mathcode:nn #1#2
  { \tex_mathcode:D #1 = \__int_eval:w #2 \__int_eval_end: }
\cs_new:Npn \char_value_mathcode:n #1
  { \tex_the:D \tex_mathcode:D \__int_eval:w #1\__int_eval_end: }
\cs_new_protected:Npn \char_show_value_mathcode:n #1
  { \tex_showthe:D \tex_mathcode:D \__int_eval:w #1 \__int_eval_end: }
\cs_new_protected:Npn \char_set_lccode:nn #1#2
  { \tex_lccode:D #1 = \__int_eval:w #2 \__int_eval_end: }
\cs_new:Npn \char_value_lccode:n #1
  { \tex_the:D \tex_lccode:D \__int_eval:w #1\__int_eval_end: }
\cs_new_protected:Npn \char_show_value_lccode:n #1
  { \tex_showthe:D \tex_lccode:D \__int_eval:w #1 \__int_eval_end: }
\cs_new_protected:Npn \char_set_uccode:nn #1#2
  { \tex_uccode:D #1 = \__int_eval:w #2 \__int_eval_end: }
\cs_new:Npn \char_value_uccode:n #1
  { \tex_the:D \tex_uccode:D \__int_eval:w #1\__int_eval_end: }
\cs_new_protected:Npn \char_show_value_uccode:n #1
  { \tex_showthe:D \tex_uccode:D \__int_eval:w #1 \__int_eval_end: }
\cs_new_protected:Npn \char_set_sfcode:nn #1#2
  { \tex_sfcode:D #1 = \__int_eval:w #2 \__int_eval_end: }
\cs_new:Npn \char_value_sfcode:n #1
  { \tex_the:D \tex_sfcode:D \__int_eval:w #1\__int_eval_end: }
\cs_new_protected:Npn \char_show_value_sfcode:n #1
  { \tex_showthe:D \tex_sfcode:D \__int_eval:w #1 \__int_eval_end: }
```

(*End definition for* `\char_set_mathcode:nn`*. This function is documented on page 53.*)

7.2 Generic tokens

\token_to_meaning:N
\token_to_meaning:c
\token_to_str:N
\token_to_str:c

These are all defined in l3basics, as they are needed "early". This is just a reminder!
(*End definition for* `\token_to_meaning:N` *and* `\token_to_meaning:c`*. These functions are documented on page* **??**.)

\token_new:Nn

Creates a new token.

```
\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 53.*)

\c_group_begin_token
\c_group_end_token
\c_math_toggle_token
\c_alignment_token
\c_parameter_token
\c_math_superscript_token
\c_math_subscript_token
\c_space_token
\c_catcode_letter_token
\c_catcode_other_token

We define these useful tokens. We have to do it by hand with the brace tokens for obvious reasons.

```
\cs_new_eq:NN \c_group_begin_token {
\cs_new_eq:NN \c_group_end_token }
\group_begin:
  \char_set_catcode_math_toggle:N \*
  \token_new:Nn \c_math_toggle_token { * }
  \char_set_catcode_alignment:N \*
  \token_new:Nn \c_alignment_token { * }
```

```
  \token_new:Nn \c_parameter_token { # }
  \token_new:Nn \c_math_superscript_token { ^ }
  \char_set_catcode_math_subscript:N \*
  \token_new:Nn \c_math_subscript_token { * }
  \token_new:Nn \c_space_token { ~ }
  \token_new:Nn \c_catcode_letter_token { a }
  \token_new:Nn \c_catcode_other_token { 1 }
\group_end:
```

(*End definition for* `\c_group_begin_token` *and others. These functions are documented on page 53.*)

`\c_catcode_active_tl`

Not an implicit token!

```
\group_begin:
  \char_set_catcode_active:N \*
  \tl_const:Nn \c_catcode_active_tl { \exp_not:N * }
\group_end:
```

(*End definition for* `\c_catcode_active_tl`*. This variable is documented on page 53.*)

`\l_char_active_seq`
`\l_char_special_seq`

Two sequences for dealing with special characters. The first is characters which may be active, and contains the active characters themselves to allow easy redefinition. The second longer list is for "special" characters more generally, and these 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`). The only complication is dealing with `_`, which requires the use of `\use:n` *and* `\use:nn`.

```
\seq_new:N \l_char_active_seq
\use:n
  {
    \group_begin:
    \char_set_catcode_active:N \"
    \char_set_catcode_active:N \$
    \char_set_catcode_active:N \&
    \char_set_catcode_active:N \^
    \char_set_catcode_active:N \_
    \char_set_catcode_active:N \~
    \use:nn
      {
        \group_end:
        \seq_set_split:Nnn \l_char_active_seq { }
      }
  }
    { { " $ & ^ _ ~ } } %$
\seq_new:N \l_char_special_seq
\seq_set_split:Nnn \l_char_special_seq { }
  { \   \" \# \$ \% \& \\ \^ \_ \{ \} \~ }
```

(*End definition for* `\l_char_active_seq` *and* `\l_char_special_seq`*. These variables are documented on page 53.*)

7.3 Token conditionals

\token_if_group_begin_p:N
\token_if_group_begin:NTF

Check if token is a begin group token. We use the constant \c_group_begin_token for this.

```
\prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_group_begin_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for \token_if_group_begin:N. These functions are documented on page 54.*)

\token_if_group_end_p:N
\token_if_group_end:NTF

Check if token is a end group token. We use the constant \c_group_end_token for this.

```
\prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_group_end_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for \token_if_group_end:N. These functions are documented on page 54.*)

\token_if_math_toggle_p:N
\token_if_math_toggle:NTF

Check if token is a math shift token. We use the constant \c_math_toggle_token for this.

```
\prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_math_toggle_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for \token_if_math_toggle:N. These functions are documented on page 54.*)

\token_if_alignment_p:N
\token_if_alignment:NTF

Check if token is an alignment tab token. We use the constant \c_alignment_token for this.

```
\prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_alignment_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for \token_if_alignment:N. These functions are documented on page 54.*)

\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.

```
\group_begin:
\cs_set_eq:NN \c_parameter_token \scan_stop:
\prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_parameter_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
\group_end:
```

(*End definition for* `\token_if_parameter:N`. *These functions are documented on page 55.*)

`\token_if_math_superscript_p:N`
`\token_if_math_superscript:NTF`

Check if token is a math superscript token. We use the constant `\c_math_superscript_token` for this.

```
\prg_new_conditional:Npnn \token_if_math_superscript:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_math_superscript_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_math_superscript:N`. *These functions are documented on page 55.*)

`\token_if_math_subscript_p:N`
`\token_if_math_subscript:NTF`

Check if token is a math subscript token. We use the constant `\c_math_subscript_token` for this.

```
\prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_math_subscript_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_math_subscript:N`. *These functions are documented on page 55.*)

`\token_if_space_p:N`
`\token_if_space:NTF`

Check if token is a space token. We use the constant `\c_space_token` for this.

```
\prg_new_conditional:Npnn \token_if_space:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_space_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_space:N`. *These functions are documented on page 55.*)

`\token_if_letter_p:N`
`\token_if_letter:NTF`

Check if token is a letter token. We use the constant `\c_catcode_letter_token` for this.

```
\prg_new_conditional:Npnn \token_if_letter:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_letter:N`. *These functions are documented on page 55.*)

`\token_if_other_p:N`
`\token_if_other:NTF`

Check if token is an other char token. We use the constant `\c_catcode_other_token` for this.

```
\prg_new_conditional:Npnn \token_if_other:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_catcode_other_token
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_other:N`. *These functions are documented on page 55.*)

\token_if_active_p:N
\token_if_active:NTF

Check if token is an active char token. We use the constant \c_catcode_active_tl for this. A technical point is that \c_catcode_active_tl is in fact a macro expanding to \exp_not:N *, where * is active.

```
\prg_new_conditional:Npnn \token_if_active:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for \token_if_active:N. These functions are documented on page 55.*)

\token_if_eq_meaning_p:NN
\token_if_eq_meaning:NNTF

Check if the tokens #1 and #2 have same meaning.

```
\prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T ,  F , TF }
  {
    \if_meaning:w  #1  #2
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for \token_if_eq_meaning:NN. These functions are documented on page 56.*)

\token_if_eq_catcode_p:NN
\token_if_eq_catcode:NNTF

Check if the tokens #1 and #2 have same category code.

```
\prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \exp_not:N #2
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for \token_if_eq_catcode:NN. These functions are documented on page 55.*)

\token_if_eq_charcode_p:NN
\token_if_eq_charcode:NNTF

Check if the tokens #1 and #2 have same character code.

```
\prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T ,  F , TF }
  {
    \if_charcode:w \exp_not:N #1 \exp_not:N #2
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for \token_if_eq_charcode:NN. These functions are documented on page 55.*)

\token_if_macro_p:N
\token_if_macro:NTF
__token_if_macro_p:w

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), and we achieve using the standard lowercasing technique.

```
\group_begin:
\char_set_catcode_other:N \M
\char_set_catcode_other:N \A
\char_set_lccode:nn { `\; } { `\: }
\char_set_lccode:nn { `\T } { `\T }
\char_set_lccode:nn { `\F } { `\F }
\tl_to_lowercase:n
  {
    \group_end:
    \prg_new_conditional:Npnn \token_if_macro:N #1 { p , T ,  F , TF }
      {
        \exp_after:wN \__token_if_macro_p:w
        \token_to_meaning:N #1 MA; \q_stop
      }
    \cs_new:Npn \__token_if_macro_p:w #1 MA #2 ; #3 \q_stop
      {
        \if_int_compare:w \pdftex_strcmp:D { #2 } { cro } = \c_zero
            \prg_return_true:
        \else:
            \prg_return_false:
        \fi:
      }
  }
```

(*End definition for* `\token_if_macro:N`*. These functions are documented on page 56.*)

\token_if_cs_p:N
\token_if_cs:N*TF*

Check if token has same catcode as a control sequence. This follows the same pattern as for `\token_if_letter:N` *etc.* We use `\scan_stop:` for this.

```
\prg_new_conditional:Npnn \token_if_cs:N #1 { p , T ,  F , TF }
  {
    \if_catcode:w \exp_not:N #1 \scan_stop:
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\token_if_cs:N`*. These functions are documented on page 56.*)

\token_if_expandable_p:N
\token_if_expandable:N*TF*

Check if token is expandable. We use the fact that TeX will temporarily convert `\exp_-not:N` ⟨*token*⟩ into `\scan_stop:` if ⟨*token*⟩ is expandable. An `undefined` token is not considered as expandable. No problem nesting the conditionals, since the third `#1` is only skipped if it is non-expandable (hence not part of TeX's conditional apparatus).

```
\prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T ,  F , TF }
  {
    \exp_after:wN \if_meaning:w \exp_not:N #1 #1
      \prg_return_false:
    \else:
      \if_cs_exist:N #1
        \prg_return_true:
      \else:
```

```
        \prg_return_false:
      \fi:
    \fi:
  }
```

(*End definition for* `\token_if_expandable:N`. *These functions are documented on page 56.*)

\token_if_chardef_p:N
\token_if_mathchardef_p:N
\token_if_dim_register_p:N
\token_if_int_register_p:N
\token_if_muskip_register_p:N
\token_if_skip_register_p:N
\token_if_toks_register_p:N
\token_if_long_macro_p:N
\token_if_protected_macro_p:N
\token_if_protected_long_macro_p:N
\token_if_chardef:NTF
\token_if_mathchardef:NTF
\token_if_dim_register:NTF
\token_if_int_register:NTF
\token_if_muskip_register:NTF
\token_if_skip_register:NTF
\token_if_toks_register:NTF
\token_if_long_macro:NTF
\token_if_protected_macro:NTF
\token_if_protected_long_macro:NTF
__token_if_chardef:w
__token_if_dim_register:w
__token_if_int_register:w
__token_if_muskip_register:w
__token_if_skip_register:w
__token_if_toks_register:w
__token_if_protected_macro:w
__token_if_long_macro:w

Most of these functions have to check the meaning of the token in question so we need to do some checkups on which characters are output by `\token_to_meaning:N`. As usual, these characters have catcode 12 so we must do some serious substitutions in the code below…

```
\group_begin:
  \char_set_lccode:nn { `T } { `T }
  \char_set_lccode:nn { `F } { `F }
  \char_set_lccode:nn { `X } { `n }
  \char_set_lccode:nn { `Y } { `t }
  \char_set_lccode:nn { `Z } { `d }
  \tl_map_inline:nn { A C E G H I K L M O P R S U X Y Z R " }
    { \char_set_catcode:nn { `#1 } \c_twelve }
```

We convert the token list to lower case and restore the catcode and lowercase code changes.

```
\tl_to_lowercase:n
  {
    \group_end:
```

First up is checking if something has been defined with `\chardef` or `\mathchardef`. This is easy since TEX thinks of such tokens as hexadecimal so it stores them as `\char"`⟨*hex number*⟩ or `\mathchar"`⟨*hex number*⟩. Grab until the first occurrence of `char"`, and compare what precedes with `\` or `\math`. In fact, the escape character may not be a backslash, so we compare with the result of converting some other control sequence to a string, namely `\char` or `\mathchar` (the auxiliary adds the `char` back).

```
    \prg_new_conditional:Npnn \token_if_chardef:N #1 { p , T ,  F , TF }
      {
        \__str_if_eq_x_return:nn
          {
            \exp_after:wN \__token_if_chardef:w
              \token_to_meaning:N #1 CHAR" \q_stop
          }
          { \token_to_str:N \char }
      }
    \prg_new_conditional:Npnn \token_if_mathchardef:N #1 { p , T ,  F , TF }
      {
        \__str_if_eq_x_return:nn
          {
            \exp_after:wN \__token_if_chardef:w
              \token_to_meaning:N #1 CHAR" \q_stop
          }
          { \token_to_str:N \mathchar }
      }
    \cs_new:Npn \__token_if_chardef:w #1 CHAR" #2 \q_stop { #1 CHAR }
```

Dim registers are a little more difficult since their `\meaning` has the form `\dimen`⟨*number*⟩, and we must take care of the two primitives `\dimen` and `\dimendef`.

```
\prg_new_conditional:Npnn \token_if_dim_register:N #1 { p , T ,  F , TF }
  {
    \if_meaning:w \tex_dimen:D #1
      \prg_return_false:
    \else:
      \if_meaning:w \tex_dimendef:D #1
        \prg_return_false:
      \else:
        \__str_if_eq_x_return:nn
          {
            \exp_after:wN \__token_if_dim_register:w
              \token_to_meaning:N #1 ZIMEX \q_stop
          }
          { \token_to_str:N \  }
      \fi:
    \fi:
  }
\cs_new:Npn \__token_if_dim_register:w #1 ZIMEX #2 \q_stop { #1 ~ }
```

Integer registers are one step harder since constants are implemented differently from variables, and we also have to take care of the primitives `\count` and `\countdef`.

```
\prg_new_conditional:Npnn \token_if_int_register:N #1 { p , T ,  F , TF }
  {
    % \token_if_chardef:NTF #1 { \prg_return_true: }
    %   {
    %     \token_if_mathchardef:NTF #1 { \prg_return_true: }
    %       {
    \if_meaning:w \tex_count:D #1
      \prg_return_false:
    \else:
      \if_meaning:w \tex_countdef:D #1
        \prg_return_false:
      \else:
        \__str_if_eq_x_return:nn
          {
            \exp_after:wN \__token_if_int_register:w
              \token_to_meaning:N #1 COUXY \q_stop
          }
          { \token_to_str:N \  }
      \fi:
    \fi:
    %       }
    %   }
  }
\cs_new:Npn \__token_if_int_register:w #1 COUXY #2 \q_stop { #1 ~ }
```

Muskip registers are done the same way as the dimension registers.

```
\prg_new_conditional:Npnn \token_if_muskip_register:N #1 { p , T ,  F , TF }
```

```
  {
    \if_meaning:w \tex_muskip:D #1
      \prg_return_false:
    \else:
      \if_meaning:w \tex_muskipdef:D #1
        \prg_return_false:
      \else:
        \__str_if_eq_x_return:nn
          {
            \exp_after:wN \__token_if_muskip_register:w
              \token_to_meaning:N #1 MUSKIP \q_stop
          }
          { \token_to_str:N \  }
      \fi:
    \fi:
  }
\cs_new:Npn \__token_if_muskip_register:w #1 MUSKIP #2 \q_stop { #1 ~ }
```

Skip registers.

```
\prg_new_conditional:Npnn \token_if_skip_register:N #1 { p , T ,  F , TF }
  {
    \if_meaning:w \tex_skip:D #1
      \prg_return_false:
    \else:
      \if_meaning:w \tex_skipdef:D #1
        \prg_return_false:
      \else:
        \__str_if_eq_x_return:nn
          {
            \exp_after:wN \__token_if_skip_register:w
              \token_to_meaning:N #1 SKIP \q_stop
          }
          { \token_to_str:N \  }
      \fi:
    \fi:
  }
\cs_new:Npn \__token_if_skip_register:w #1 SKIP #2 \q_stop { #1 ~ }
```

Toks registers.

```
\prg_new_conditional:Npnn \token_if_toks_register:N #1 { p , T ,  F , TF }
  {
    \if_meaning:w \tex_toks:D #1
      \prg_return_false:
    \else:
      \if_meaning:w \tex_toksdef:D #1
        \prg_return_false:
      \else:
        \__str_if_eq_x_return:nn
          {
            \exp_after:wN \__token_if_toks_register:w
              \token_to_meaning:N #1 YOKS \q_stop
```

```
            }
            { \token_to_str:N \  }
        \fi:
      \fi:
    }
  \cs_new:Npn \__token_if_toks_register:w #1 YOKS #2 \q_stop { #1 ~ }
```

Protected macros.

```
\prg_new_conditional:Npnn \token_if_protected_macro:N #1
  { p , T ,  F , TF }
  {
    \__str_if_eq_x_return:nn
      {
        \exp_after:wN \__token_if_protected_macro:w
          \token_to_meaning:N #1 PROYECYEZ~MACRO \q_stop
      }
      { \token_to_str:N \  }
  }
\cs_new:Npn \__token_if_protected_macro:w
  #1 PROYECYEZ~MACRO #2 \q_stop { #1 ~ }
```

Long macros and protected long macros share an auxiliary.

```
\prg_new_conditional:Npnn \token_if_long_macro:N #1 { p , T ,  F , TF }
  {
    \__str_if_eq_x_return:nn
      {
        \exp_after:wN \__token_if_long_macro:w
          \token_to_meaning:N #1 LOXG~MACRO \q_stop
      }
      { \token_to_str:N \  }
  }
\prg_new_conditional:Npnn \token_if_protected_long_macro:N #1
  { p , T ,  F , TF }
  {
    \__str_if_eq_x_return:nn
      {
        \exp_after:wN \__token_if_long_macro:w
          \token_to_meaning:N #1 LOXG~MACRO \q_stop
      }
      { \token_to_str:N \protected \token_to_str:N \  }
  }
\cs_new:Npn \__token_if_long_macro:w #1 LOXG~MACRO #2 \q_stop { #1 ~ }
```

Finally the `\tl_to_lowercase:n` ends!

```
}
```

(*End definition for `\token_if_chardef:N` and others. These functions are documented on page 56.*)

\token_if_primitive_p:N
\token_if_primitive:NTF
__token_if_primitive:NNw
__token_if_primitive_space:w
__token_if_primitive_nullfont:N
__token_if_primitive_loop:N
__token_if_primitive:Nw
__token_if_primitive_undefined:N

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 exceptions remain: `\tex_undefined:D`, which is not a primitive, and `\nullfont`, which is a primitive.

Spaces cannot be grabbed in an undelimited way, so we check them separately. If there is a space, we test for `\nullfont`. Otherwise, we go through characters one by one, and stop at the first character less than `‘A` (this is not quite a test for "only letters", but is close enough to work in this context). If this first character is `:` then we have a primitive, or `\tex_undefined:D`, and if it is `"` or a digit, then the token is not a primitive.

```
\tex_chardef:D \c_token_A_int = ‘A ~ %
\group_begin:
\char_set_catcode_other:N \;
\char_set_lccode:nn { ‘\; } { ‘\: }
\char_set_lccode:nn { ‘\T } { ‘\T }
\char_set_lccode:nn { ‘\F } { ‘\F }
\tl_to_lowercase:n {
  \group_end:
  \prg_new_conditional:Npnn \token_if_primitive:N #1 { p , T , F , TF }
    {
      \token_if_macro:NTF #1
        \prg_return_false:
        {
          \exp_after:wN \__token_if_primitive:NNw
          \token_to_meaning:N #1 ; ; ; \q_stop #1
        }
    }
  \cs_new:Npn \__token_if_primitive:NNw #1#2 #3 ; #4 \q_stop
    {
      \tl_if_empty:oTF { \__token_if_primitive_space:w #3 ~ }
        { \__token_if_primitive_loop:N #3 ; \q_stop }
        { \__token_if_primitive_nullfont:N }
    }
}
\cs_new:Npn \__token_if_primitive_space:w #1 ~ { }
\cs_new:Npn \__token_if_primitive_nullfont:N #1
  {
    \if_meaning:w \tex_nullfont:D #1
```

```
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \__token_if_primitive_loop:N #1
  {
    \if_int_compare:w `#1 < \c_token_A_int %
      \exp_after:wN \__token_if_primitive:Nw
      \exp_after:wN #1
    \else:
      \exp_after:wN \__token_if_primitive_loop:N
    \fi:
  }
\cs_new:Npn \__token_if_primitive:Nw #1 #2 \q_stop
  {
    \if:w : #1
      \exp_after:wN \__token_if_primitive_undefined:N
    \else:
      \prg_return_false:
      \exp_after:wN \use_none:n
    \fi:
  }
\cs_new:Npn \__token_if_primitive_undefined:N #1
  {
    \if_cs_exist:N #1
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\token_if_primitive:N`. *These functions are documented on page 57.*)

7.4 Peeking ahead at the next token

```
⟨@@=peek⟩
```

Peeking ahead is implemented using a two part mechanism. The outer level provides a defined interface to the lower level material. This allows a large amount of code to be shared. There are four cases:

1. peek at the next token;
2. peek at the next non-space token;
3. peek at the next token and remove it;
4. peek at the next non-space token and remove it.

\l_peek_token \g_peek_token

Storage tokens which are publicly documented: the token peeked.

```
\cs_new_eq:NN \l_peek_token ?
\cs_new_eq:NN \g_peek_token ?
```

(*End definition for* `\l_peek_token`. *This variable is documented on page 58.*)

`\l__peek_search_token` The token to search for as an implicit token: *cf.* `\l__peek_search_tl`.

```
\cs_new_eq:NN \l__peek_search_token ?
```

(*End definition for* `\l__peek_search_token`. *This variable is documented on page ??.*)

`\l__peek_search_tl` The token to search for as an explicit token: *cf.* `\l__peek_search_token`.

```
\tl_new:N \l__peek_search_tl
```

(*End definition for* `\l__peek_search_tl`. *This variable is documented on page ??.*)

`\__peek_true:w`
`\__peek_true_aux:w`
`\__peek_false:w`
`\__peek_tmp:w`

Functions used by the branching and space-stripping code.

```
\cs_new_nopar:Npn \__peek_true:w  { }
\cs_new_nopar:Npn \__peek_true_aux:w  { }
\cs_new_nopar:Npn \__peek_false:w { }
\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.

```
\cs_new_protected_nopar:Npn \peek_after:Nw
  { \tex_futurelet:D \l_peek_token }
\cs_new_protected_nopar:Npn \peek_gafter:Nw
  { \tex_global:D \tex_futurelet:D \g_peek_token }
```

(*End definition for* `\peek_after:Nw`. *This function is documented on page 58.*)

`\__peek_true_remove:w` A function to remove the next token and then regain control.

```
\cs_new_protected:Npn \__peek_true_remove:w
  {
    \group_align_safe_end:
    \tex_afterassignment:D \__peek_true_aux:w
    \cs_set_eq:NN \__peek_tmp:w
  }
```

(*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.

```
\cs_new_protected:Npn \__peek_token_generic:NNTF #1#2#3#4
  {
    \cs_set_eq:NN \l__peek_search_token #2
    \tl_set:Nn \l__peek_search_tl {#2}
    \cs_set_nopar:Npx \__peek_true:w
      {
        \exp_not:N \group_align_safe_end:
        \exp_not:n {#3}
      }
    \cs_set_nopar:Npx \__peek_false:w
      {
        \exp_not:N \group_align_safe_end:
```

```
        \exp_not:n {#4}
      }
    \group_align_safe_begin:
      \peek_after:Nw #1
  }
\cs_new_protected:Npn \__peek_token_generic:NNT #1#2#3
  { \__peek_token_generic:NNTF #1 #2 {#3} { } }
\cs_new_protected:Npn \__peek_token_generic:NNF #1#2#3
  { \__peek_token_generic:NNTF #1 #2 { } {#3} }
```

(*End definition for* `\__peek_token_generic:NNTF`*. This function is documented on page* **??**.)

__peek_token_remove_generic:NNTF

For token removal there needs to be a call to the auxiliary function which does the work.

```
\cs_new_protected:Npn \__peek_token_remove_generic:NNTF #1#2#3#4
  {
    \cs_set_eq:NN \l__peek_search_token #2
    \tl_set:Nn \l__peek_search_tl {#2}
    \cs_set_eq:NN \__peek_true:w \__peek_true_remove:w
    \cs_set_nopar:Npx \__peek_true_aux:w { \exp_not:n {#3} }
    \cs_set_nopar:Npx \__peek_false:w
      {
        \exp_not:N \group_align_safe_end:
        \exp_not:n {#4}
      }
    \group_align_safe_begin:
      \peek_after:Nw #1
  }
\cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
  { \__peek_token_remove_generic:NNTF #1 #2 {#3} { } }
\cs_new_protected:Npn \__peek_token_remove_generic:NNF #1#2#3
  { \__peek_token_remove_generic:NNTF #1 #2 { } {#3} }
```

(*End definition for* `\__peek_token_remove_generic:NNTF`*. This function is documented on page* **??**.)

__peek_execute_branches_meaning:

The meaning test is straight forward.

```
\cs_new_nopar:Npn \__peek_execute_branches_meaning:
  {
    \if_meaning:w \l_peek_token \l__peek_search_token
      \exp_after:wN \__peek_true:w
    \else:
      \exp_after:wN \__peek_false:w
    \fi:
  }
```

(*End definition for* `\__peek_execute_branches_meaning:`*. This function is documented on page* **??**.)

__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.

```
\cs_new_nopar:Npn \__peek_execute_branches_catcode:
  { \if_catcode:w \__peek_execute_branches_catcode_aux: }
\cs_new_nopar:Npn \__peek_execute_branches_charcode:
  { \if_charcode:w \__peek_execute_branches_catcode_aux: }
\cs_new_nopar:Npn \__peek_execute_branches_catcode_aux:
  {
        \if_catcode:w \exp_not:N \l_peek_token \scan_stop:
          \exp_after:wN \exp_after:wN
          \exp_after:wN \__peek_execute_branches_catcode_auxii:N
          \exp_after:wN \exp_not:N
        \else:
          \exp_after:wN \__peek_execute_branches_catcode_auxiii:
        \fi:
  }
\cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1
  {
        \exp_not:N #1
        \exp_after:wN \exp_not:N \l__peek_search_tl
      \exp_after:wN \__peek_true:w
    \else:
      \exp_after:wN \__peek_false:w
    \fi:
    #1
  }
\cs_new_nopar:Npn \__peek_execute_branches_catcode_auxiii:
  {
        \exp_not:N \l_peek_token
        \exp_after:wN \exp_not:N \l__peek_search_tl
      \exp_after:wN \__peek_true:w
    \else:
      \exp_after:wN \__peek_false:w
    \fi:
  }
```

(*End definition for* `\__peek_execute_branches_catcode:` *and* `\__peek_execute_branches_charcode:`. *These functions are documented on page* **??**.)

__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: `\tex_romannumeral:D -‘0` removes one space.

```
\cs_new_protected_nopar:Npn \__peek_ignore_spaces_execute_branches:
  {
    \if_meaning:w \l_peek_token \c_space_token
      \exp_after:wN \peek_after:Nw
      \exp_after:wN \__peek_ignore_spaces_execute_branches:
      \tex_romannumeral:D -‘0
    \else:
      \exp_after:wN \__peek_execute_branches:
    \fi:
  }
```

(*End definition for* `\__peek_ignore_spaces_execute_branches:`. *This function is documented on page* ??.)

__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.

```
\group_begin:
  \cs_set:Npn \__peek_def:nnnn #1#2#3#4
    {
      \__peek_def:nnnnn {#1} {#2} {#3} {#4} { TF }
      \__peek_def:nnnnn {#1} {#2} {#3} {#4} { T }
      \__peek_def:nnnnn {#1} {#2} {#3} {#4} { F }
    }
  \cs_set:Npn \__peek_def:nnnnn #1#2#3#4#5
    {
      \cs_new_protected_nopar:cpx { #1 #5 }
        {
          \tl_if_empty:nF {#2}
            { \exp_not:n { \cs_set_eq:NN \__peek_execute_branches: #2 } }
          \exp_not:c { #3 #5 }
          \exp_not:n {#4}
        }
    }
```

(*End definition for* `\__peek_def:nnnn`.)

\peek_catcode:NTF
\peek_catcode_ignore_spaces:NTF
\peek_catcode_remove:NTF
\peek_catcode_remove_ignore_spaces:NTF

With everything in place the definitions can take place. First for category codes.

```
  \__peek_def:nnnn { peek_catcode:N }
    { }
    { __peek_token_generic:NN }
    { \__peek_execute_branches_catcode: }
  \__peek_def:nnnn { peek_catcode_ignore_spaces:N }
    { \__peek_execute_branches_catcode: }
    { __peek_token_generic:NN }
    { \__peek_ignore_spaces_execute_branches: }
```

```
  \__peek_def:nnnn { peek_catcode_remove:N }
    { }
    { __peek_token_remove_generic:NN }
    { \__peek_execute_branches_catcode: }
  \__peek_def:nnnn { peek_catcode_remove_ignore_spaces:N }
    { \__peek_execute_branches_catcode: }
    { __peek_token_remove_generic:NN }
    { \__peek_ignore_spaces_execute_branches: }
```

(*End definition for* `\peek_catcode:NTF` *and others. These functions are documented on page 59.*)

\peek_charcode:NTF
\peek_charcode_ignore_spaces:NTF
\peek_charcode_remove:NTF
\peek_charcode_remove_ignore_spaces:NTF

Then for character codes.

```
  \__peek_def:nnnn { peek_charcode:N }
    { }
    { __peek_token_generic:NN }
    { \__peek_execute_branches_charcode: }
  \__peek_def:nnnn { peek_charcode_ignore_spaces:N }
    { \__peek_execute_branches_charcode: }
    { __peek_token_generic:NN }
    { \__peek_ignore_spaces_execute_branches: }
  \__peek_def:nnnn { peek_charcode_remove:N }
    { }
    { __peek_token_remove_generic:NN }
    { \__peek_execute_branches_charcode: }
  \__peek_def:nnnn { peek_charcode_remove_ignore_spaces:N }
    { \__peek_execute_branches_charcode: }
    { __peek_token_remove_generic:NN }
    { \__peek_ignore_spaces_execute_branches: }
```

(*End definition for* `\peek_charcode:NTF` *and others. These functions are documented on page 60.*)

\peek_meaning:NTF
\peek_meaning_ignore_spaces:NTF
\peek_meaning_remove:NTF
\peek_meaning_remove_ignore_spaces:NTF

Finally for meaning, with the group closed to remove the temporary definition functions.

```
  \__peek_def:nnnn { peek_meaning:N }
    { }
    { __peek_token_generic:NN }
    { \__peek_execute_branches_meaning: }
  \__peek_def:nnnn { peek_meaning_ignore_spaces:N }
    { \__peek_execute_branches_meaning: }
    { __peek_token_generic:NN }
    { \__peek_ignore_spaces_execute_branches: }
  \__peek_def:nnnn { peek_meaning_remove:N }
    { }
    { __peek_token_remove_generic:NN }
    { \__peek_execute_branches_meaning: }
  \__peek_def:nnnn { peek_meaning_remove_ignore_spaces:N }
    { \__peek_execute_branches_meaning: }
    { __peek_token_remove_generic:NN }
    { \__peek_ignore_spaces_execute_branches: }
\group_end:
```

(*End definition for* `\peek_meaning:NTF` *and others. These functions are documented on page 60.*)

7.5 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.

```
\exp_args:Nno \use:nn
  { \cs_new:Npn \__peek_get_prefix_arg_replacement:wN #1 }
  { \tl_to_str:n { macro : } #2 -> #3 \q_stop #4 }
  { #4 {#1} {#2} {#3} }
\cs_new:Npn \token_get_prefix_spec:N #1
  {
    \token_if_macro:NTF #1
      {
        \exp_after:wN \__peek_get_prefix_arg_replacement:wN
          \token_to_meaning:N #1 \q_stop \use_i:nnn
      }
      { \scan_stop: }
  }
\cs_new:Npn \token_get_arg_spec:N #1
  {
    \token_if_macro:NTF #1
      {
        \exp_after:wN \__peek_get_prefix_arg_replacement:wN
          \token_to_meaning:N #1 \q_stop \use_ii:nnn
      }
      { \scan_stop: }
  }
\cs_new:Npn \token_get_replacement_spec:N #1
  {
    \token_if_macro:NTF #1
      {
        \exp_after:wN \__peek_get_prefix_arg_replacement:wN
          \token_to_meaning:N #1 \q_stop \use_iii:nnn
      }
      { \scan_stop: }
  }
```

(*End definition for* `\token_get_prefix_spec:N`. *This function is documented on page 61.*)

```
⟨/initex | package⟩
```

8 l3int implementation

```
⟨*initex | package⟩
⟨@@=int⟩
```

The following test files are used for this code: m3int001,m3int002,m3int03.

__int_to_roman:w
\if_int_compare:w

Done in l3basics.

(*End definition for* `\__int_to_roman:w`. *This function is documented on page 74.*)

__int_value:w
__int_eval:w
__int_eval_end:
\if_int_odd:w
\if_case:w

Here are the remaining primitives for number comparisons and expressions.

```
\cs_new_eq:NN \__int_value:w      \tex_number:D
\cs_new_eq:NN \__int_eval:w       \etex_numexpr:D
\cs_new_eq:NN \__int_eval_end:    \tex_relax:D
\cs_new_eq:NN \if_int_odd:w     \tex_ifodd:D
\cs_new_eq:NN \if_case:w        \tex_ifcase:D
```

(*End definition for* `\__int_value:w`. *This function is documented on page 74.*)

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.

```
⟨*initex⟩
\cs_set:Npn \int_eval:n #1 { \__int_value:w \__int_eval:w #1 \__int_eval_end: }
⟨/initex⟩
⟨*package⟩
\cs_new:Npn \int_eval:n #1 { \__int_value:w \__int_eval:w #1 \__int_eval_end: }
⟨/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.

```
\cs_new:Npn \int_abs:n #1
  {
    \__int_value:w \exp_after:wN \__int_abs:N
      \int_use:N \__int_eval:w #1 \__int_eval_end:
    \exp_stop_f:
  }
\cs_new:Npn \__int_abs:N #1
  { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
\cs_set:Npn \int_max:nn #1#2
  {
    \__int_value:w \exp_after:wN \__int_maxmin:wwN
      \int_use:N \__int_eval:w #1 \exp_after:wN ;
      \int_use:N \__int_eval:w #2 ;
      >
    \exp_stop_f:
  }
\cs_set:Npn \int_min:nn #1#2
  {
    \__int_value:w \exp_after:wN \__int_maxmin:wwN
      \int_use:N \__int_eval:w #1 \exp_after:wN ;
      \int_use:N \__int_eval:w #2 ;
      <
```

```
    \exp_stop_f:
  }
\cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
  {
    \if_int_compare:w #1 #3 #2 ~
      #1
    \else:
      #2
    \fi:
  }
```

(*End definition for* `\int_abs:n`. *This function is documented on page 63.*)

`\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.

```
\cs_new:Npn \int_div_truncate:nn #1#2
  {
    \int_use:N \__int_eval:w
      \exp_after:wN \__int_div_truncate:NwNw
      \int_use:N \__int_eval:w #1 \exp_after:wN ;
      \int_use:N \__int_eval:w #2 ;
    \__int_eval_end:
  }
\cs_new:Npn \__int_div_truncate:NwNw #1#2; #3#4;
  {
    \if_meaning:w 0 #1
      \c_zero
    \else:
      (
        #1#2
        \if_meaning:w - #1 + \else: - \fi:
        ( \if_meaning:w - #3 - \fi: #3#4 - \c_one ) / \c_two
      )
    \fi:
    / #3#4
  }
```

For the sake of completeness:

```
\cs_new:Npn \int_div_round:nn #1#2
  { \__int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }
```

Finally there's the modulus operation.

```
\cs_new:Npn \int_mod:nn #1#2
```

```
  {
    \__int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
      \__int_value:w \__int_eval:w #1 \exp_after:wN ;
      \__int_value:w \__int_eval:w #2 ;
    \__int_eval_end:
  }
\cs_new:Npn \__int_mod:ww #1; #2;
  { #1 - ( \__int_div_truncate:NwNw #1 ; #2 ; ) * #2 }
```

(*End definition for* `\int_div_truncate:nn`*. This function is 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.)

```
⟨*package⟩
\cs_new_protected:Npn \int_new:N #1
  {
    \__chk_if_free_cs:N #1
    \cs:w newcount \cs_end: #1
  }
⟨/package⟩
\cs_generate_variant:Nn \int_new:N { c }
```

(*End definition for* `\int_new:N` *and* `\int_new:c`*. These functions are documented on page ??.*)

`\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.

```
\cs_new_protected:Npn \int_const:Nn #1#2
  {
    \int_compare:nNnTF {#2} > \c_minus_one
      {
        \int_compare:nNnTF {#2} > \c__max_constdef_int
          {
            \int_new:N #1
            \int_gset:Nn #1 {#2}
          }
          {
            \__chk_if_free_cs:N #1
            \tex_global:D \__int_constdef:Nw #1 =
              \__int_eval:w #2 \__int_eval_end:
          }
      }
      {
        \int_new:N #1
        \int_gset:Nn #1 {#2}
      }
  }
```

```
\cs_generate_variant:Nn \int_const:Nn { c }
\pdftex_if_engine:TF
  {
    \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
    \tex_mathchardef:D \c__max_constdef_int 32 767 ~
  }
  {
    \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D
    \tex_chardef:D \c__max_constdef_int 1 114 111 ~
  }
```

(*End definition for* `\int_const:Nn` *and* `\int_const:cn`. *These functions are documented on page* **??**.)

`\int_zero:N`
`\int_zero:c`
`\int_gzero:N`
`\int_gzero:c`

Functions that reset an ⟨*integer*⟩ register to zero.

```
\cs_new_protected:Npn \int_zero:N  #1 { #1 = \c_zero }
\cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero }
\cs_generate_variant:Nn \int_zero:N  { c }
\cs_generate_variant:Nn \int_gzero:N { c }
```

(*End definition for* `\int_zero:N` *and* `\int_zero:c`. *These functions are documented on page* **??**.)

`\int_zero_new:N`
`\int_zero_new:c`
`\int_gzero_new:N`
`\int_gzero_new:c`

Create a register if needed, otherwise clear it.

```
\cs_new_protected:Npn \int_zero_new:N  #1
  { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }
\cs_new_protected:Npn \int_gzero_new:N #1
  { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }
\cs_generate_variant:Nn \int_zero_new:N  { c }
\cs_generate_variant:Nn \int_gzero_new:N { c }
```

(*End definition for* `\int_zero_new:N` *and others. These functions are documented on page* **??**.)

`\int_set_eq:NN`
`\int_set_eq:cN`
`\int_set_eq:Nc`
`\int_set_eq:cc`
`\int_gset_eq:NN`
`\int_gset_eq:cN`
`\int_gset_eq:Nc`
`\int_gset_eq:cc`

Setting equal means using one integer inside the set function of another.

```
\cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }
\cs_generate_variant:Nn \int_set_eq:NN {      c }
\cs_generate_variant:Nn \int_set_eq:NN { Nc , cc }
\cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\cs_generate_variant:Nn \int_gset_eq:NN {      c }
\cs_generate_variant:Nn \int_gset_eq:NN { Nc , cc }
```

(*End definition for* `\int_set_eq:NN` *and others. These functions are documented on page* **??**.)

`\int_if_exist_p:N`
`\int_if_exist_p:c`
`\int_if_exist:NTF`
`\int_if_exist:cTF`

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \int_if_exist:N \cs_if_exist:N { TF , T , F , p }
\prg_new_eq_conditional:NNn \int_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(*End definition for* `\int_if_exist:N` *and* `\int_if_exist:c`. *These functions are documented on page* **??**.)

8.3 Setting and incrementing integers

\int_add:Nn
\int_add:cn
\int_gadd:Nn
\int_gadd:cn
\int_sub:Nn
\int_sub:cn
\int_gsub:Nn
\int_gsub:cn

Adding and subtracting to and from a counter …

```
\cs_new_protected:Npn \int_add:Nn #1#2
  { \tex_advance:D #1 by \__int_eval:w #2 \__int_eval_end: }
\cs_new_protected:Npn \int_sub:Nn #1#2
  { \tex_advance:D #1 by - \__int_eval:w #2 \__int_eval_end: }
\cs_new_protected_nopar:Npn \int_gadd:Nn
  { \tex_global:D \int_add:Nn }
\cs_new_protected_nopar:Npn \int_gsub:Nn
  { \tex_global:D \int_sub:Nn }
\cs_generate_variant:Nn \int_add:Nn  { c }
\cs_generate_variant:Nn \int_gadd:Nn { c }
\cs_generate_variant:Nn \int_sub:Nn  { c }
\cs_generate_variant:Nn \int_gsub:Nn { c }
```

(*End definition for* `\int_add:Nn` *and* `\int_add:cn`*. These functions are documented on page* **??**.)

\int_incr:N
\int_incr:c
\int_gincr:N
\int_gincr:c
\int_decr:N
\int_decr:c
\int_gdecr:N
\int_gdecr:c

Incrementing and decrementing of integer registers is done with the following functions.

```
\cs_new_protected:Npn \int_incr:N #1
  { \tex_advance:D #1 \c_one }
\cs_new_protected:Npn \int_decr:N #1
  { \tex_advance:D #1 \c_minus_one }
\cs_new_protected_nopar:Npn \int_gincr:N
  { \tex_global:D \int_incr:N }
\cs_new_protected_nopar:Npn \int_gdecr:N
  { \tex_global:D \int_decr:N }
\cs_generate_variant:Nn \int_incr:N  { c }
\cs_generate_variant:Nn \int_decr:N  { c }
\cs_generate_variant:Nn \int_gincr:N { c }
\cs_generate_variant:Nn \int_gdecr:N { c }
```

(*End definition for* `\int_incr:N` *and* `\int_incr:c`*. These functions are documented on page* **??**.)

\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.

```
\cs_new_protected:Npn \int_set:Nn #1#2
  { #1 ~ \__int_eval:w #2\__int_eval_end: }
\cs_new_protected_nopar:Npn \int_gset:Nn { \tex_global:D \int_set:Nn }
\cs_generate_variant:Nn \int_set:Nn  { c }
\cs_generate_variant:Nn \int_gset:Nn { c }
```

(*End definition for* `\int_set:Nn` *and* `\int_set:cn`*. These functions are documented on page* **??**.)

8.4 Using integers

\int_use:N
\int_use:c

Here is how counters are accessed:

```
\cs_new_eq:NN \int_use:N \tex_the:D
\cs_new:Npn \int_use:c #1 { \int_use:N \cs:w #1 \cs_end: }
```

(*End definition for* `\int_use:N` *and* `\int_use:c`*. These functions are documented on page* **??**.)

8.5 Integer expression conditionals

__prg_compare_error:
__prg_compare_error:NNw

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.

```
\cs_new_protected_nopar:Npn \__prg_compare_error:
  {
    \if_int_compare:w \c_zero \c_zero \fi:
    =
    \__prg_compare_error:
  }
\cs_new:Npn \__prg_compare_error:Nw
    #1#2 \q_stop
  {
    { }
    \c_zero \fi:
    \__msg_kernel_expandable_error:nnn
      { kernel } { unknown-comparison } {#1}
    \prg_return_false:
  }
```

(*End definition for* __prg_compare_error: *and* __prg_compare_error:NNw.)

\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.

```
\prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
  {
    \exp_after:wN \__int_compare:w
    \int_use:N \__int_eval:w #1 \__prg_compare_error:
  }
\cs_new:Npn \__int_compare:w #1 \__prg_compare_error:
  {
    \exp_after:wN \if_false: \__int_value:w
      \__int_compare:Nw #1 e { = nd_ } \q_stop
  }
```

The goal here is to find an ⟨*operand*⟩ and a ⟨*comparison*⟩. The ⟨*operand*⟩ is already evaluated, but we cannot yet grab it as an argument. To access the following relation symbol, we remove the number by applying __int_to_roman:w, after making sure that the argument becomes non-positive: its roman numeral representation is then empty. Then probe the first two tokens with __int_compare:NNw to determine the relation symbol, building a control sequence from it. 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.

```
\cs_new:Npn \__int_compare:Nw #1#2 \q_stop
  {
    \exp_after:wN \__int_compare:NNw
      \__int_to_roman:w - 0 #2 \q_mark
    #1#2 \q_stop
  }
\cs_new:Npn \__int_compare:NNw #1#2#3 \q_mark
  {
    \etex_unexpanded:D
    \use:c { __int_compare_  #1  \if_meaning:w = #2 =  \fi: :NNw }
      \__prg_compare_error:Nw #1
  }
```

When the last ⟨*operand*⟩ is seen, __int_compare:NNw receives e and =nd_ as arguments, hence calling __int_compare_end_=:NNw to end the loop: return the result of the last comparison (involving the operand that we just found). When a normal relation is found, the appropriate auxiliary calls __int_compare:nnN where #1 is \if_int_compare:w or \reverse_if:N \if_int_compare:w, #2 is the ⟨*operand*⟩, and #3 is one of <, =, or >. As announced earlier, we leave the ⟨*operand*⟩ for the previous conditional. If this conditional is true the result of the test is known, so we remove all tokens and return false. Otherwise, we apply the conditional #1 to the ⟨*operand*⟩ #2 and the comparison #3, and call __int_compare:Nw to look for additional operands, after evaluating the following expression.

```
\cs_new:cpn { __int_compare_end_=:NNw } #1#2#3 e #4 \q_stop
  {
    {#3} \exp_stop_f:
```

```
    \prg_return_false: \else: \prg_return_true: \fi:
  }
\cs_new:Npn \__int_compare:nnN #1#2#3
  {
        {#2} \exp_stop_f:
      \prg_return_false: \exp_after:wN \use_none_delimit_by_q_stop:w
    \fi:
    #1 #2 #3 \exp_after:wN \__int_compare:Nw \__int_value:w \__int_eval:w
  }
```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding `\__prg_compare_error:Nw` ⟨*token*⟩ responsible for error detection.

```
\cs_new:cpn { __int_compare_=:NNw } #1#2#3 =
  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
\cs_new:cpn { __int_compare_<:NNw } #1#2#3 <
  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
\cs_new:cpn { __int_compare_>:NNw } #1#2#3 >
  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }
\cs_new:cpn { __int_compare_==:NNw } #1#2#3 ==
  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
\cs_new:cpn { __int_compare_!=:NNw } #1#2#3 !=
  { \__int_compare:nnN { \if_int_compare:w } {#3} = }
\cs_new:cpn { __int_compare_<=:NNw } #1#2#3 <=
  { \__int_compare:nnN { \if_int_compare:w } {#3} > }
\cs_new:cpn { __int_compare_>=:NNw } #1#2#3 >=
  { \__int_compare:nnN { \if_int_compare:w } {#3} < }
```

(*End definition for* `\int_compare:n`. *These functions are documented on page 66.*)

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

More efficient but less natural in typing.

```
\prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
  {
    \if_int_compare:w \__int_eval:w #1 #2 \__int_eval:w #3 \__int_eval_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\int_compare:nNn`. *These functions are documented on page 65.*)

`\int_case:nn`
`\int_case:nnTF`
`\__int_case:nnTF`
`\__int_case:nw`
`\__int_case_end:nw`

For integer cases, the first task to fully expand the check condition. The over all idea is then much the same as for `\str_case:nn(TF)` as described in l3basics.

```
\cs_new:Npn \int_case:nnTF #1
  {
    \tex_romannumeral:D
    \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
  }
\cs_new:Npn \int_case:nnT #1#2#3
  {
```

```
    \tex_romannumeral:D
    \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
  }
\cs_new:Npn \int_case:nnF #1#2
  {
    \tex_romannumeral:D
    \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
  }
\cs_new:Npn \int_case:nn #1#2
  {
    \tex_romannumeral:D
    \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
  }
\cs_new:Npn \__int_case:nnTF #1#2#3#4
  { \__int_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
\cs_new:Npn \__int_case:nw #1#2#3
  {
    \int_compare:nNnTF {#1} = {#2}
      { \__int_case_end:nw {#3} }
      { \__int_case:nw {#1} }
  }
\cs_new_eq:NN \__int_case_end:nw \__prg_case_end:nw
```

(*End definition for* `\int_case:nn`. *This function is documented on page 67.*)

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

A predicate function.

```
\prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF}
  {
    \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
  {
    \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
      \prg_return_false:
    \else:
      \prg_return_true:
    \fi:
  }
```

(*End definition for* `\int_if_odd:n`. *These functions are documented on page 67.*)

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.

```
\cs_new:Npn \int_while_do:nn #1#2
  {
```

```
    \int_compare:nT {#1}
      {
        #2
        \int_while_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \int_until_do:nn #1#2
  {
    \int_compare:nF {#1}
      {
        #2
        \int_until_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \int_do_while:nn #1#2
  {
    #2
    \int_compare:nT {#1}
      { \int_do_while:nn {#1} {#2} }
  }
\cs_new:Npn \int_do_until:nn #1#2
  {
    #2
    \int_compare:nF {#1}
      { \int_do_until:nn {#1} {#2} }
  }
```

(*End definition for* `\int_while_do:nn`*. This function is documented on page* 68*.*)

`\int_while_do:nNnn`
`\int_until_do:nNnn`
`\int_do_while:nNnn`
`\int_do_until:nNnn`

As above but not using the more natural syntax.

```
\cs_new:Npn \int_while_do:nNnn #1#2#3#4
  {
    \int_compare:nNnT {#1} #2 {#3}
      {
        #4
        \int_while_do:nNnn {#1} #2 {#3} {#4}
      }
  }
\cs_new:Npn \int_until_do:nNnn #1#2#3#4
  {
    \int_compare:nNnF {#1} #2 {#3}
      {
        #4
        \int_until_do:nNnn {#1} #2 {#3} {#4}
      }
  }
\cs_new:Npn \int_do_while:nNnn #1#2#3#4
  {
    #4
    \int_compare:nNnT {#1} #2 {#3}
      { \int_do_while:nNnn {#1} #2 {#3} {#4} }
```

```
  }
\cs_new:Npn \int_do_until:nNnn #1#2#3#4
  {
    #4
    \int_compare:nNnF {#1} #2 {#3}
      { \int_do_until:nNnn {#1} #2 {#3} {#4} }
  }
```

(*End definition for* `\int_while_do:nNnn`*. This function is documented on page 67.*)

8.7 Integer step functions

`\int_step_function:nnnN`
`\__int_step:NnnnN`

Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```
\cs_new:Npn \int_step_function:nnnN #1#2#3#4
  {
    \int_compare:nNnTF {#2} > \c_zero
      { \exp_args:NNf \__int_step:NnnnN > }
      {
        \int_compare:nNnTF {#2} = \c_zero
          {
            \__msg_kernel_expandable_error:nnn { kernel } { zero-step } {#4}
            \use_none:nnnn
          }
          { \exp_args:NNf \__int_step:NnnnN < }
      }
      { \int_eval:n {#1} } {#2} {#3} #4
  }
\cs_new:Npn \__int_step:NnnnN #1#2#3#4#5
  {
    \int_compare:nNnF {#2} #1 {#4}
      {
        #5 {#2}
        \exp_args:NNf \__int_step:NnnnN
          #1 { \int_eval:n { #2 + #3 } } {#3} {#4} #5
      }
  }
```

(*End definition for* `\int_step_function:nnnN`*. This function is documented on page 69.*)

`\int_step_inline:nnnn`
`\int_step_variable:nnnNn`
`\__int_step:NNnnnn`

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\int_step_function:nnnN`. We put a `\__prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g__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.

```
\cs_new_protected_nopar:Npn \int_step_inline:nnnn
  {
```

```
    \int_gincr:N \g__prg_map_int
    \exp_args:NNc \__int_step:NNnnnn
      \cs_gset_nopar:Npn
      { __prg_map_ \int_use:N \g__prg_map_int :w }
  }
\cs_new_protected:Npn \int_step_variable:nnnNn #1#2#3#4#5
  {
    \int_gincr:N \g__prg_map_int
    \exp_args:NNc \__int_step:NNnnnn
      \cs_gset_nopar:Npx
      { __prg_map_ \int_use:N \g__prg_map_int :w }
      {#1}{#2}{#3}
      {
        \tl_set:Nn \exp_not:N #4 {##1}
        \exp_not:n {#5}
      }
  }
\cs_new_protected:Npn \__int_step:NNnnnn #1#2#3#4#5#6
  {
    #1 #2 ##1 {#6}
    \int_step_function:nnnN {#3} {#4} {#5} #2
    \__prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__prg_map_int }
  }
```

(*End definition for* `\int_step_inline:nnnn`. *This function is documented on page 69.*)

8.8 Formatting integers

`\int_to_arabic:n`

Nothing exciting here.

```
\cs_new:Npn \int_to_arabic:n #1 { \int_eval:n {#1} }
```

(*End definition for* `\int_to_arabic:n`. *This function is documented on page 69.*)

`\int_to_symbols:nnn`
`\__int_to_symbols:nnnn`

For conversion of integers to arbitrary symbols the method is in general as follows. The input number (`#1`) is compared to the total number of symbols available at each place (`#2`). If the input is larger than the total number of symbols available then the modulus is needed, with one added so that the positions don't have to number from zero. Using an `f`-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a 'nice' number at each stage. Of course, if the initial input was small enough then there is no problem and everything is easy.

```
\cs_new:Npn \int_to_symbols:nnn #1#2#3
  {
    \int_compare:nNnTF {#1} > {#2}
      {
        \exp_args:NNo \exp_args:No \__int_to_symbols:nnnn
          {
            \int_case:nn
              { 1 + \int_mod:nn { #1 - 1 } {#2} }
              {#3}
          }
```

```
          {#1} {#2} {#3}
      }
      { \int_case:nn {#1} {#3} }
  }
\cs_new:Npn \__int_to_symbols:nnnn #1#2#3#4
  {
    \exp_args:Nf \int_to_symbols:nnn
      { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}
    #1
  }
```

(*End definition for* `\int_to_symbols:nnn`. *This function is documented on page 70.*)

`\int_to_alph:n`
`\int_to_Alph:n`

These both use the above function with input functions that make sense for the alphabet in English.

```
\cs_new:Npn \int_to_alph:n #1
  {
    \int_to_symbols:nnn {#1} { 26 }
      {
        {  1 } { a }
        {  2 } { b }
        {  3 } { c }
        {  4 } { d }
        {  5 } { e }
        {  6 } { f }
        {  7 } { g }
        {  8 } { h }
        {  9 } { i }
        { 10 } { j }
        { 11 } { k }
        { 12 } { l }
        { 13 } { m }
        { 14 } { n }
        { 15 } { o }
        { 16 } { p }
        { 17 } { q }
        { 18 } { r }
        { 19 } { s }
        { 20 } { t }
        { 21 } { u }
        { 22 } { v }
        { 23 } { w }
        { 24 } { x }
        { 25 } { y }
        { 26 } { z }
      }
  }
\cs_new:Npn \int_to_Alph:n #1
  {
    \int_to_symbols:nnn {#1} { 26 }
```

```
      {
        {  1 } { A }
        {  2 } { B }
        {  3 } { C }
        {  4 } { D }
        {  5 } { E }
        {  6 } { F }
        {  7 } { G }
        {  8 } { H }
        {  9 } { I }
        { 10 } { J }
        { 11 } { K }
        { 12 } { L }
        { 13 } { M }
        { 14 } { N }
        { 15 } { O }
        { 16 } { P }
        { 17 } { Q }
        { 18 } { R }
        { 19 } { S }
        { 20 } { T }
        { 21 } { U }
        { 22 } { V }
        { 23 } { W }
        { 24 } { X }
        { 25 } { Y }
        { 26 } { Z }
      }
  }
```

(*End definition for* `\int_to_alph:n` *and* `\int_to_Alph:n`*. These functions are documented on page 70.*)

`\int_to_base:nn`
`\__int_to_base:nn`
`\__int_to_Base:nn`
`\__int_to_base:nnN`
`\__int_to_Base:nnN`
`\__int_to_base:nnnN`
`\__int_to_Base:nnnN`
`\__int_to_letter:n`
`\__int_to_Letter:n`

Converting from base ten (`#1`) to a second base (`#2`) starts with computing `#1`: if it is a complicated calculation, we shouldn't perform it twice. Then check the sign, store it, either `-` or `\c_empty_tl`, and feed the absolute value to the next auxiliary function.

```
\cs_new:Npn \int_to_base:nn #1
  { \exp_args:Nf \__int_to_base:nn { \int_eval:n {#1} } }
\cs_new:Npn \int_to_Base:nn #1
  { \exp_args:Nf \__int_to_Base:nn { \int_eval:n {#1} } }
\cs_new:Npn \__int_to_base:nn #1#2
  {
    \int_compare:nNnTF {#1} < \c_zero
      { \exp_args:No \__int_to_base:nnN { \use_none:n #1 } {#2} - }
      { \__int_to_base:nnN {#1} {#2} \c_empty_tl }
  }
\cs_new:Npn \__int_to_Base:nn #1#2
  {
    \int_compare:nNnTF {#1} < \c_zero
      { \exp_args:No \__int_to_Base:nnN { \use_none:n #1 } {#2} - }
      { \__int_to_Base:nnN {#1} {#2} \c_empty_tl }
```

```
  }
```

Here, the idea is to provide a recursive system to deal with the input. The output is built up after the end of the function. At each pass, the value in `#1` is checked to see if it is less than the new base (`#2`). If it is, then it is converted directly, putting the sign back in front. On the other hand, if the value to convert is greater than or equal to the new base then the modulus and remainder values are found. The modulus is converted to a symbol and put on the right, and the remainder is carried forward to the next round.

```
\cs_new:Npn \__int_to_base:nnN #1#2#3
  {
    \int_compare:nNnTF {#1} < {#2}
      { \exp_last_unbraced:Nf #3 { \__int_to_letter:n {#1} } }
      {
        \exp_args:Nf \__int_to_base:nnnN
          { \__int_to_letter:n { \int_mod:nn {#1} {#2} } }
          {#1}
          {#2}
          #3
      }
  }
\cs_new:Npn \__int_to_base:nnnN #1#2#3#4
  {
    \exp_args:Nf \__int_to_base:nnN
      { \int_div_truncate:nn {#2} {#3} }
      {#3}
      #4
    #1
  }
\cs_new:Npn \__int_to_Base:nnN #1#2#3
  {
    \int_compare:nNnTF {#1} < {#2}
      { \exp_last_unbraced:Nf #3 { \__int_to_Letter:n {#1} } }
      {
        \exp_args:Nf \__int_to_Base:nnnN
          { \__int_to_Letter:n { \int_mod:nn {#1} {#2} } }
          {#1}
          {#2}
          #3
      }
  }
\cs_new:Npn \__int_to_Base:nnnN #1#2#3#4
  {
    \exp_args:Nf \__int_to_Base:nnN
      { \int_div_truncate:nn {#2} {#3} }
      {#3}
      #4
    #1
  }
```

Convert to a letter only if necessary, otherwise simply return the value unchanged. It would be cleaner to use `\int_case:nn`, but in our case, the cases are contiguous, so it is forty times faster to use the `\if_case:w` primitive. The first `\exp_after:wN` expands the conditional, jumping to the correct case, the second one expands after the resulting character to close the conditional. Since `#1` might be an expression, and not directly a single digit, we need to evaluate it properly, and expand the trailing `\fi:`.

```
\cs_new:Npn \__int_to_letter:n #1
  {
    \exp_after:wN \exp_after:wN
    \if_case:w \__int_eval:w #1 - \c_ten \__int_eval_end:
         a
    \or: b
    \or: c
    \or: d
    \or: e
    \or: f
    \or: g
    \or: h
    \or: i
    \or: j
    \or: k
    \or: l
    \or: m
    \or: n
    \or: o
    \or: p
    \or: q
    \or: r
    \or: s
    \or: t
    \or: u
    \or: v
    \or: w
    \or: x
    \or: y
    \or: z
    \else: \__int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
    \fi:
  }
\cs_new:Npn \__int_to_Letter:n #1
  {
    \exp_after:wN \exp_after:wN
    \if_case:w \__int_eval:w #1 - \c_ten \__int_eval_end:
         A
    \or: B
    \or: C
    \or: D
    \or: E
    \or: F
    \or: G
```

```
    \or: H
    \or: I
    \or: J
    \or: K
    \or: L
    \or: M
    \or: N
    \or: O
    \or: P
    \or: Q
    \or: R
    \or: S
    \or: T
    \or: U
    \or: V
    \or: W
    \or: X
    \or: Y
    \or: Z
    \else: \__int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
    \fi:
  }
```

(*End definition for* `\int_to_base:nn`*. This function is documented on page 71.*)

`\int_to_bin:n`
`\int_to_hex:n`
`\int_to_Hex:n`
`\int_to_oct:n`

Wrappers around the generic function.

```
\cs_new:Npn \int_to_bin:n #1
  { \int_to_base:nn {#1} { 2 } }
\cs_new:Npn \int_to_hex:n #1
  { \int_to_base:nn {#1} { 16 } }
\cs_new:Npn \int_to_Hex:n #1
  { \int_to_Base:nn {#1} { 16 } }
\cs_new:Npn \int_to_oct:n #1
  { \int_to_base:nn {#1} { 8 } }
```

(*End definition for* `\int_to_bin:n` *and others. These functions are documented on page 71.*)

`\int_to_roman:n`
`\int_to_Roman:n`
`\__int_to_roman:N`
`\__int_to_roman:N`
`\__int_to_roman_i:w`
`\__int_to_roman_v:w`
`\__int_to_roman_x:w`
`\__int_to_roman_l:w`
`\__int_to_roman_c:w`
`\__int_to_roman_d:w`
`\__int_to_roman_m:w`
`\__int_to_roman_Q:w`
`\__int_to_Roman_i:w`
`\__int_to_Roman_v:w`
`\__int_to_Roman_x:w`
`\__int_to_Roman_l:w`
`\__int_to_Roman_c:w`
`\__int_to_Roman_d:w`
`\__int_to_Roman_m:w`
`\__int_to_Roman_Q:w`

The `\__int_to_roman:w` primitive creates tokens of category code 12 (other). Usually, what is actually wanted is letters. The approach here is to convert the output of the primitive into letters using appropriate control sequence names. That keeps everything expandable. The loop will be terminated by the conversion of the `Q`.

```
\cs_new:Npn \int_to_roman:n #1
  {
    \exp_after:wN \__int_to_roman:N
      \__int_to_roman:w \int_eval:n {#1} Q
  }
\cs_new:Npn \__int_to_roman:N #1
  {
    \use:c { __int_to_roman_ #1 :w }
    \__int_to_roman:N
  }
```

```
\cs_new:Npn \int_to_Roman:n #1
  {
    \exp_after:wN \__int_to_Roman_aux:N
      \__int_to_roman:w \int_eval:n {#1} Q
  }
\cs_new:Npn \__int_to_Roman_aux:N #1
  {
    \use:c { __int_to_Roman_ #1 :w }
    \__int_to_Roman_aux:N
  }
\cs_new_nopar:Npn \__int_to_roman_i:w { i }
\cs_new_nopar:Npn \__int_to_roman_v:w { v }
\cs_new_nopar:Npn \__int_to_roman_x:w { x }
\cs_new_nopar:Npn \__int_to_roman_l:w { l }
\cs_new_nopar:Npn \__int_to_roman_c:w { c }
\cs_new_nopar:Npn \__int_to_roman_d:w { d }
\cs_new_nopar:Npn \__int_to_roman_m:w { m }
\cs_new_nopar:Npn \__int_to_roman_Q:w #1 { }
\cs_new_nopar:Npn \__int_to_Roman_i:w { I }
\cs_new_nopar:Npn \__int_to_Roman_v:w { V }
\cs_new_nopar:Npn \__int_to_Roman_x:w { X }
\cs_new_nopar:Npn \__int_to_Roman_l:w { L }
\cs_new_nopar:Npn \__int_to_Roman_c:w { C }
\cs_new_nopar:Npn \__int_to_Roman_d:w { D }
\cs_new_nopar:Npn \__int_to_Roman_m:w { M }
\cs_new:Npn \__int_to_Roman_Q:w #1 { }
```

(*End definition for* `\int_to_roman:n` *and* `\int_to_Roman:n`*. These functions are documented on page* *71*.)

8.9 Converting from other formats to integers

__int_get_sign:n
__int_get_digits:n
__int_get_sign_and_digits:nNNN
__int_get_sign_and_digits:oNNN

Finding a number and its sign requires dealing with an arbitrary list of + and - symbols. This is done by working through token by token until there is something else at the start of the input. The sign of the input is tracked by the first Boolean used by the auxiliary function.

```
\cs_new:Npn \__int_get_sign:n #1
  {
    \__int_get_sign_and_digits:nNNN {#1}
      \c_true_bool \c_true_bool \c_false_bool
  }
\cs_new:Npn \__int_get_digits:n #1
  {
    \__int_get_sign_and_digits:nNNN {#1}
      \c_true_bool \c_false_bool \c_true_bool
  }
```

The auxiliary loops through, finding sign tokens and removing them. The sign itself is carried through as a flag.

```
\cs_new:Npn \__int_get_sign_and_digits:nNNN #1#2#3#4
```

```
  {
    \exp_args:Nf \tl_if_head_eq_charcode:nNTF {#1} -
      {
        \bool_if:NTF #2
          {
            \__int_get_sign_and_digits:oNNN
              { \use_none:n #1 } \c_false_bool #3#4
          }
          {
            \__int_get_sign_and_digits:oNNN
              { \use_none:n #1 } \c_true_bool #3#4
          }
      }
      {
        \exp_args:Nf \tl_if_head_eq_charcode:nNTF {#1} +
          { \__int_get_sign_and_digits:oNNN { \use_none:n #1 } #2#3#4 }
          {
            \bool_if:NT #3 { \bool_if:NF #2 - }
            \bool_if:NT #4 {#1}
          }
      }
  }
\cs_generate_variant:Nn \__int_get_sign_and_digits:nNNN { o }
```

(*End definition for* `\__int_get_sign:n`.)

`\int_from_alph:n`
`\__int_from_alph:n`
`\__int_from_alph:nN`
`\__int_from_alph:N`

The aim here is to iterate through the input, converting one letter at a time to a number. The same approach is also used for base conversion, but this needs a different final auxiliary.

```
\cs_new:Npn \int_from_alph:n #1
  {
    \int_eval:n
      {
        \__int_get_sign:n {#1}
        \exp_args:Nf \__int_from_alph:n { \__int_get_digits:n {#1} }
      }
  }
\cs_new:Npn \__int_from_alph:n #1
  { \__int_from_alph:nN { 0 } #1 \q_nil }
\cs_new:Npn \__int_from_alph:nN #1#2
  {
    \quark_if_nil:NTF #2
      {#1}
      {
        \exp_args:Nf \__int_from_alph:nN
          { \int_eval:n { #1 * 26 + \__int_from_alph:N #2 } }
      }
  }
\cs_new:Npn \__int_from_alph:N #1
  { \int_eval:n { `#1 - \int_compare:nNnTF { `#1 } < { 91 } { 64 } { 96 } } }
```

(*End definition for* `\int_from_alph:n`. *This function is documented on page* 71.)

`\int_from_base:nn`
`\__int_from_base:nn`
`\__int_from_base:nnN`
`\__int_from_base:N`

Conversion to base ten means stripping off the sign then iterating through the input one token at a time. The total number is then added up as the code loops.

```
\cs_new:Npn \int_from_base:nn #1#2
  {
    \int_eval:n
      {
        \__int_get_sign:n {#1}
        \exp_args:Nf \__int_from_base:nn
          { \__int_get_digits:n {#1} } {#2}
      }
  }
\cs_new:Npn \__int_from_base:nn #1#2
  { \__int_from_base:nnN { 0 } { #2 } #1 \q_nil }
\cs_new:Npn \__int_from_base:nnN #1#2#3
  {
    \quark_if_nil:NTF #3
      {#1}
      {
        \exp_args:Nf \__int_from_base:nnN
          { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
          {#2}
      }
  }
```

The conversion here will take lower or upper case letters and turn them into the appropriate number, hence the two-part nature of the function.

```
\cs_new:Npn \__int_from_base:N #1
  {
    \int_compare:nNnTF { `#1 } < { 58 }
      {#1}
      {
        \int_eval:n
          { `#1 - \int_compare:nNnTF { `#1 } < { 91 } { 55 } { 87 } }
      }
  }
```

(*End definition for* `\int_from_base:nn`. *This function is documented on page* 72.)

`\int_from_bin:n`
`\int_from_hex:n`
`\int_from_oct:n`

Wrappers around the generic function.

```
\cs_new:Npn \int_from_bin:n #1
  { \int_from_base:nn {#1} \c_two }
\cs_new:Npn \int_from_hex:n #1
  { \int_from_base:nn {#1} \c_sixteen }
\cs_new:Npn \int_from_oct:n #1
  { \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* 72.)

\c__int_from_roman_i_int
\c__int_from_roman_v_int
\c__int_from_roman_x_int
\c__int_from_roman_l_int
\c__int_from_roman_c_int
\c__int_from_roman_d_int
\c__int_from_roman_m_int
\c__int_from_roman_I_int
\c__int_from_roman_V_int
\c__int_from_roman_X_int
\c__int_from_roman_L_int
\c__int_from_roman_C_int
\c__int_from_roman_D_int
\c__int_from_roman_M_int

Constants used to convert from Roman numerals to integers.

```
\int_const:cn { c__int_from_roman_i_int } { 1 }
\int_const:cn { c__int_from_roman_v_int } { 5 }
\int_const:cn { c__int_from_roman_x_int } { 10 }
\int_const:cn { c__int_from_roman_l_int } { 50 }
\int_const:cn { c__int_from_roman_c_int } { 100 }
\int_const:cn { c__int_from_roman_d_int } { 500 }
\int_const:cn { c__int_from_roman_m_int } { 1000 }
\int_const:cn { c__int_from_roman_I_int } { 1 }
\int_const:cn { c__int_from_roman_V_int } { 5 }
\int_const:cn { c__int_from_roman_X_int } { 10 }
\int_const:cn { c__int_from_roman_L_int } { 50 }
\int_const:cn { c__int_from_roman_C_int } { 100 }
\int_const:cn { c__int_from_roman_D_int } { 500 }
\int_const:cn { c__int_from_roman_M_int } { 1000 }
```

(*End definition for \c__int_from_roman_i_int and others. These variables are documented on page ??.*)

\int_from_roman:n
__int_from_roman:NN
__int_from_roman_end:w
__int_from_roman_clean_up: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.

```
\cs_new:Npn \int_from_roman:n #1
  {
    \tl_if_blank:nF {#1}
      {
        \exp_after:wN \__int_from_roman_end:w
          \__int_value:w \__int_eval:w
            \__int_from_roman:NN #1 Q \q_stop
      }
  }
\cs_new:Npn \__int_from_roman:NN #1#2
  {
    \str_if_eq:nnTF {#1} { Q }
      {#1#2}
      {
        \str_if_eq:nnTF {#2} { Q }
          {
            \int_if_exist:cF { c__int_from_roman_ #1 _int }
              { \__int_from_roman_clean_up:w }
            +
            \use:c { c__int_from_roman_ #1 _int }
            #2
          }
          {
            \int_if_exist:cF { c__int_from_roman_ #1 _int }
              { \__int_from_roman_clean_up:w }
            \int_if_exist:cF { c__int_from_roman_ #2 _int }
              { \__int_from_roman_clean_up:w }
            \int_compare:nNnTF
              { \use:c { c__int_from_roman_ #1 _int } }
```

```
                <
                { \use:c { c__int_from_roman_ #2 _int  } }
                {
                  + \use:c { c__int_from_roman_ #2 _int }
                  - \use:c { c__int_from_roman_ #1 _int }
                  \__int_from_roman:NN
                }
                {
                  + \use:c { c__int_from_roman_ #1 _int }
                  \__int_from_roman:NN #2
                }
            }
        }
  }
\cs_new:Npn \__int_from_roman_end:w #1 Q #2 \q_stop
  { \tl_if_empty:nTF {#2} {#1} {#2} }
\cs_new:Npn \__int_from_roman_clean_up:w #1 Q { + 0 Q -1 }
```

(*End definition for* `\int_from_roman:n`*. This function is documented on page 72.*)

8.10 Viewing integer

`\int_show:N`
`\int_show:c`

```
\cs_new_eq:NN \int_show:N \__kernel_register_show:N
\cs_new_eq:NN \int_show:c \__kernel_register_show:c
```

(*End definition for* `\int_show:N` *and* `\int_show:c`*. These functions are documented on page* ??.)

`\int_show:n` We don't use the TeX primitive `\showthe` to show integer expressions: this gives a more unified output, since the closing brace is read by the integer expression in all cases.

```
\cs_new_protected:Npn \int_show:n #1
  { \etex_showtokens:D \exp_after:wN { \int_use:N \__int_eval:w #1 } }
```

(*End definition for* `\int_show:n`*. This function is documented on page 72.*)

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 73.*)

`\c_zero` Again, one in l3basics for obvious reasons.

(*End definition for* `\c_zero`*. This variable is documented on page 73.*)

`\c_six`
`\c_seven`
Once again, in l3basics.

(*End definition for* `\c_six` *and* `\c_seven`*. These variables are documented on page 73.*)

`\c_twelve`
`\c_sixteen`
`\c_one`
`\c_two`
`\c_three`
`\c_four`
`\c_five`
`\c_eight`
`\c_nine`
`\c_ten`
`\c_eleven`
`\c_thirteen`
`\c_fourteen`
`\c_fifteen`

Low-number values not previously defined.

```
\int_const:Nn \c_one      {  1 }
\int_const:Nn \c_two      {  2 }
\int_const:Nn \c_three    {  3 }
\int_const:Nn \c_four     {  4 }
\int_const:Nn \c_five     {  5 }
```

```
\int_const:Nn \c_eight    {  8 }
\int_const:Nn \c_nine     {  9 }
\int_const:Nn \c_ten      { 10 }
\int_const:Nn \c_eleven   { 11 }
\int_const:Nn \c_thirteen { 13 }
\int_const:Nn \c_fourteen { 14 }
\int_const:Nn \c_fifteen  { 15 }
```

(*End definition for* `\c_one` *and others. These variables are documented on page 73.*)

`\c_thirty_two`

One middling value.

```
\int_const:Nn \c_thirty_two { 32 }
```

(*End definition for* `\c_thirty_two`*. This variable is documented on page 73.*)

`\c_two_hundred_fifty_five`
`\c_two_hundred_fifty_six`

Two classic mid-range integer constants.

```
\int_const:Nn \c_two_hundred_fifty_five { 255 }
\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 73.*)

`\c_one_hundred`
`\c_one_thousand`
`\c_ten_thousand`

Simple runs of powers of ten.

```
\int_const:Nn \c_one_hundred  {   100 }
\int_const:Nn \c_one_thousand {  1000 }
\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 73.*)

`\c_max_int`

The largest number allowed is $2^{31} - 1$

```
\int_const:Nn \c_max_int { 2 147 483 647 }
```

(*End definition for* `\c_max_int`*. This variable is documented on page 73.*)

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.

```
\int_new:N \l_tmpa_int
\int_new:N \l_tmpb_int
\int_new:N \g_tmpa_int
\int_new:N \g_tmpb_int
```

(*End definition for* `\l_tmpa_int` *and* `\l_tmpb_int`*. These variables are documented on page 73.*)

8.13 Deprecated functions

\int_case:nnn

Deprecated 2013-07-15.

```
\cs_new_eq:NN \int_case:nnn \int_case:nnF
```

(*End definition for* \int_case:nnn. *This function is documented on page* **??**.)

\int_to_binary:n
\int_from_binary:n
\int_to_hexadecimal:n
\int_from_hexadecimal:n
\int_to_octal:n
\int_from_octal:n

Deprecated 2014-02-11.

```
\cs_new_eq:NN \int_to_binary:n      \int_to_bin:n
\cs_new_eq:NN \int_to_hexadecimal:n \int_to_Hex:n
\cs_new_eq:NN \int_to_octal:n       \int_to_oct:n
\cs_new_eq:NN \int_from_binary:n      \int_from_bin:n
\cs_new_eq:NN \int_from_hexadecimal:n \int_from_hex:n
\cs_new_eq:NN \int_from_octal:n       \int_from_oct:n
```

(*End definition for* \int_to_binary:n *and* \int_from_binary:n. *These functions are documented on page* **??**.)

```
⟨/initex | package⟩
```

9 l3skip implementation

```
⟨*initex | package⟩
⟨@@=dim⟩
```

9.1 Length primitives renamed

\if_dim:w
__dim_eval:w
__dim_eval_end:

Primitives renamed.

```
\cs_new_eq:NN \if_dim:w      \tex_ifdim:D
\cs_new_eq:NN \__dim_eval:w      \etex_dimexpr:D
\cs_new_eq:NN \__dim_eval_end:   \tex_relax:D
```

(*End definition for* \if_dim:w. *This function is documented on page 89*.)

9.2 Creating and initialising dim variables

\dim_new:N
\dim_new:c

Allocating ⟨*dim*⟩ registers . . .

```
⟨*package⟩
\cs_new_protected:Npn \dim_new:N #1
  {
    \__chk_if_free_cs:N #1
    \cs:w newdimen \cs_end: #1
  }
⟨/package⟩
\cs_generate_variant:Nn \dim_new:N { c }
```

(*End definition for* \dim_new:N *and* \dim_new:c. *These functions are documented on page* **??**.)

\dim_const:Nn
\dim_const:cn

Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
\cs_new_protected:Npn \dim_const:Nn #1
  {
    \dim_new:N #1
    \dim_gset:Nn #1
  }
\cs_generate_variant:Nn \dim_const:Nn { c }
```

(*End definition for* `\dim_const:Nn` *and* `\dim_const:cn`*. These functions are documented on page* **??**.)

\dim_zero:N
\dim_zero:c
\dim_gzero:N
\dim_gzero:c

Reset the register to zero.

```
\cs_new_protected:Npn \dim_zero:N #1 { #1 \c_zero_dim }
\cs_new_protected:Npn \dim_gzero:N { \tex_global:D \dim_zero:N }
\cs_generate_variant:Nn \dim_zero:N  { c }
\cs_generate_variant:Nn \dim_gzero:N { c }
```

(*End definition for* `\dim_zero:N` *and* `\dim_zero:c`*. These functions are documented on page* **??**.)

\dim_zero_new:N
\dim_zero_new:c
\dim_gzero_new:N
\dim_gzero_new:c

Create a register if needed, otherwise clear it.

```
\cs_new_protected:Npn \dim_zero_new:N  #1
  { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
\cs_new_protected:Npn \dim_gzero_new:N #1
  { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
\cs_generate_variant:Nn \dim_zero_new:N  { c }
\cs_generate_variant:Nn \dim_gzero_new:N { c }
```

(*End definition for* `\dim_zero_new:N` *and others. These functions are documented on page* **??**.)

\dim_if_exist_p:N
\dim_if_exist_p:c
\dim_if_exist:NTF
\dim_if_exist:cTF

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N { TF , T , F , p }
\prg_new_eq_conditional:NNn \dim_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(*End definition for* `\dim_if_exist:N` *and* `\dim_if_exist:c`*. These functions are documented on page* **??**.)

9.3 Setting dim variables

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

Setting dimensions is easy enough.

```
\cs_new_protected:Npn \dim_set:Nn #1#2
  { #1 ~ \__dim_eval:w #2 \__dim_eval_end: }
\cs_new_protected:Npn \dim_gset:Nn { \tex_global:D \dim_set:Nn }
\cs_generate_variant:Nn \dim_set:Nn  { c }
\cs_generate_variant:Nn \dim_gset:Nn { c }
```

(*End definition for* `\dim_set:Nn` *and* `\dim_set:cn`*. These functions are documented on page* **??**.)

\dim_set_eq:NN
\dim_set_eq:cN
\dim_set_eq:Nc
\dim_set_eq:cc
\dim_gset_eq:NN
\dim_gset_eq:cN
\dim_gset_eq:Nc
\dim_gset_eq:cc

All straightforward.

```
\cs_new_protected:Npn \dim_set_eq:NN #1#2 { #1 = #2 }
\cs_generate_variant:Nn \dim_set_eq:NN {      c }
\cs_generate_variant:Nn \dim_set_eq:NN { Nc , cc }
\cs_new_protected:Npn \dim_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\cs_generate_variant:Nn \dim_gset_eq:NN {      c }
\cs_generate_variant:Nn \dim_gset_eq:NN { Nc , cc }
```

(*End definition for* `\dim_set_eq:NN` *and others. These functions are documented on page* **??**.)

\dim_add:Nn
\dim_add:cn
\dim_gadd:Nn
\dim_gadd:cn
\dim_sub:Nn
\dim_sub:cn
\dim_gsub:Nn
\dim_gsub:cn

Using `by` here deals with the (incorrect) case `\dimen123`.

```
\cs_new_protected:Npn \dim_add:Nn #1#2
  { \tex_advance:D #1 by \__dim_eval:w #2 \__dim_eval_end: }
\cs_new_protected:Npn \dim_gadd:Nn { \tex_global:D \dim_add:Nn }
\cs_generate_variant:Nn \dim_add:Nn  { c }
\cs_generate_variant:Nn \dim_gadd:Nn { c }
\cs_new_protected:Npn \dim_sub:Nn #1#2
  { \tex_advance:D #1 by - \__dim_eval:w #2 \__dim_eval_end: }
\cs_new_protected:Npn \dim_gsub:Nn { \tex_global:D \dim_sub:Nn }
\cs_generate_variant:Nn \dim_sub:Nn  { c }
\cs_generate_variant:Nn \dim_gsub:Nn { c }
```

(*End definition for* `\dim_add:Nn` *and* `\dim_add:cn`*. These functions are documented on page* **??**.)

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.

```
\cs_new:Npn \dim_abs:n #1
  {
    \exp_after:wN \__dim_abs:N
    \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
  }
\cs_new:Npn \__dim_abs:N #1
  { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
\cs_set:Npn \dim_max:nn #1#2
  {
    \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
      \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
      \dim_use:N \__dim_eval:w #2 ;
      >
    \__dim_eval_end:
  }
\cs_set:Npn \dim_min:nn #1#2
  {
    \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
      \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
      \dim_use:N \__dim_eval:w #2 ;
      <
    \__dim_eval_end:
  }
\cs_new:Npn \__dim_maxmin:wwN #1 ; #2 ; #3
  {
    \if_dim:w #1 #3 #2 ~
      #1
    \else:
      #2
    \fi:
```

```
  }
```

(*End definition for* `\dim_abs:n`. *This function is documented on page 77.*)

`\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.

```
\cs_new:Npn \dim_ratio:nn #1#2
  { \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
\cs_new:Npn \__dim_ratio:n #1
  { \__int_value:w \__dim_eval:w #1 \__dim_eval_end: }
```

(*End definition for* `\dim_ratio:nn`. *This function is documented on page 78.*)

9.5 Dimension expression conditionals

`\dim_compare_p:nNn` `\dim_compare:nNnTF`

Simple comparison.

```
\prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
  {
    \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\dim_compare:nNn`. *These functions are documented on page 78.*)

`\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 >.

```
\prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
  {
    \exp_after:wN \__dim_compare:w
    \dim_use:N \__dim_eval:w #1 \__prg_compare_error:
  }
\cs_new:Npn \__dim_compare:w #1 \__prg_compare_error:
  {
    \exp_after:wN \if_false: \tex_romannumeral:D -`0
      \__dim_compare:wNN #1 ? { = \__dim_compare_end:w \else: } \q_stop
  }
\exp_args:Nno \use:nn
  { \cs_new:Npn \__dim_compare:wNN #1 }
  { \tl_to_str:n {pt} }
  #2#3
  {
      \if_meaning:w = #3
        \use:c { __dim_compare_#2:w }
      \fi:
        #1 pt \exp_stop_f:
```

```
      \prg_return_false:
      \exp_after:wN \use_none_delimit_by_q_stop:w
    \fi:
    \reverse_if:N \if_dim:w #1 pt #2
      \exp_after:wN \__dim_compare:wNN
      \dim_use:N \__dim_eval:w #3
  }
\cs_new:cpn { __dim_compare_ ! :w }
    #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
\cs_new:cpn { __dim_compare_ = :w }
    #1 \__dim_eval:w = { #1 \__dim_eval:w }
\cs_new:cpn { __dim_compare_ < :w }
    #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
\cs_new:cpn { __dim_compare_ > :w }
    #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
\cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \q_stop
  { #1 \prg_return_false: \else: \prg_return_true: \fi: }
```

(*End definition for* `\dim_compare:n`*. These functions are documented on page* *79*.)

`\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.

```
\cs_new:Npn \dim_case:nnTF #1
  {
    \tex_romannumeral:D
    \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} }
  }
\cs_new:Npn \dim_case:nnT #1#2#3
  {
    \tex_romannumeral:D
    \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { }
  }
\cs_new:Npn \dim_case:nnF #1#2
  {
    \tex_romannumeral:D
    \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { }
  }
\cs_new:Npn \dim_case:nn #1#2
  {
    \tex_romannumeral:D
    \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { }
  }
\cs_new:Npn \__dim_case:nnTF #1#2#3#4
  { \__dim_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
\cs_new:Npn \__dim_case:nw #1#2#3
  {
    \dim_compare:nNnTF {#1} = {#2}
      { \__dim_case_end:nw {#3} }
      { \__dim_case:nw {#1} }
  }
\cs_new_eq:NN \__dim_case_end:nw \__prg_case_end:nw
```

(*End definition for* `\dim_case:nn`*. This function is documented on page 80.*)

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.

```
\cs_set:Npn \dim_while_do:nn #1#2
  {
    \dim_compare:nT {#1}
      {
        #2
        \dim_while_do:nn {#1} {#2}
      }
  }
\cs_set:Npn \dim_until_do:nn #1#2
  {
    \dim_compare:nF {#1}
      {
        #2
        \dim_until_do:nn {#1} {#2}
      }
  }
\cs_set:Npn \dim_do_while:nn #1#2
  {
    #2
    \dim_compare:nT {#1}
      { \dim_do_while:nn {#1} {#2} }
  }
\cs_set:Npn \dim_do_until:nn #1#2
  {
    #2
    \dim_compare:nF {#1}
      { \dim_do_until:nn {#1} {#2} }
  }
```

(*End definition for* `\dim_while_do:nn`*. This function is documented on page 81.*)

\dim_while_do:nNnn
\dim_until_do:nNnn
\dim_do_while:nNnn
\dim_do_until:nNnn

`while_do` and `do_while` functions for dimensions. Same as for the `int` type only the names have changed.

```
\cs_set:Npn \dim_while_do:nNnn #1#2#3#4
  {
    \dim_compare:nNnT {#1} #2 {#3}
      {
        #4
        \dim_while_do:nNnn {#1} #2 {#3} {#4}
      }
  }
\cs_set:Npn \dim_until_do:nNnn #1#2#3#4
  {
  \dim_compare:nNnF {#1} #2 {#3}
```

```
      {
        #4
        \dim_until_do:nNnn {#1} #2 {#3} {#4}
      }
  }
\cs_set:Npn \dim_do_while:nNnn #1#2#3#4
  {
    #4
    \dim_compare:nNnT {#1} #2 {#3}
      { \dim_do_while:nNnn {#1} #2 {#3} {#4} }
  }
\cs_set:Npn \dim_do_until:nNnn #1#2#3#4
  {
    #4
    \dim_compare:nNnF {#1} #2 {#3}
      { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
  }
```

(*End definition for* `\dim_while_do:nNnn`. *This function is documented on page 80.*)

9.7 Using dim expressions and variables

`\dim_eval:n`

Evaluating a dimension expression expandably.

```
\cs_new:Npn \dim_eval:n #1
  { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }
```

(*End definition for* `\dim_eval:n`. *This function is documented on page 81.*)

`\__dim_strip_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.

```
\cs_new:Npn \__dim_strip_bp:n #1
  {
    \__dim_strip_pt:n
      {
        \__dim_eval:w ( #1 ) * 800 / 803 \__dim_eval_end:
      }
  }
```

(*End definition for* `\__dim_strip_bp:n`.)

`\__dim_strip_pt:n`
`\__dim_strip_pt:w`

A function which comes up often enough to deserve a place in the kernel. The idea here is that the input is assumed to be in `pt`, but can be given in other units, while the output is the value of the dimension in `pt` but with no units given. This is used a lot by low-level manipulations.

```
\cs_new:Npn \__dim_strip_pt:n #1
  {
    \exp_after:wN
      \__dim_strip_pt:w \dim_use:N \__dim_eval:w #1 \__dim_eval_end: \q_stop
  }
\use:x
  {
```

```
    \cs_new:Npn \exp_not:N \__dim_strip_pt:w
      ##1 . ##2 \tl_to_str:n { pt } ##3 \exp_not:N \q_stop
      {
        ##1
        \exp_not:N \int_compare:nNnT {##2} > \c_zero
          { . ##2 }
      }
  }
```

(*End definition for* `\__dim_strip_pt:n`*. This function is documented on page 89.*)

\dim_use:N
\dim_use:c

Accessing a ⟨*dim*⟩.

```
\cs_new_eq:NN \dim_use:N \tex_the:D
\cs_generate_variant:Nn \dim_use:N { c }
```

(*End definition for* `\dim_use:N` *and* `\dim_use:c`*. These functions are documented on page* **??**.)

9.8 Viewing dim variables

\dim_show:N
\dim_show:c

Diagnostics.

```
\cs_new_eq:NN  \dim_show:N \__kernel_register_show:N
\cs_generate_variant:Nn \dim_show:N { c }
```

(*End definition for* `\dim_show:N` *and* `\dim_show:c`*. These functions are documented on page* **??**.)

\dim_show:n

Diagnostics. We don't use the TeX primitive `\showthe` to show dimension expressions: this gives a more unified output, since the closing brace is read by the dimension expression in all cases.

```
\cs_new_protected:Npn \dim_show:n #1
  { \etex_showtokens:D \exp_after:wN { \dim_use:N \__dim_eval:w #1 } }
```

(*End definition for* `\dim_show:n`*. This function is documented on page 82.*)

9.9 Constant dimensions

\c_zero_dim
\c_max_dim

Constant dimensions: in package mode, a couple of registers can be saved.

```
\dim_const:Nn \c_zero_dim { 0 pt }
\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
\l_tmpb_dim
\g_tmpa_dim
\g_tmpb_dim

We provide two local and two global scratch registers, maybe we need more or less.

```
\dim_new:N \l_tmpa_dim
\dim_new:N \l_tmpb_dim
\dim_new:N \g_tmpa_dim
\dim_new:N \g_tmpb_dim
```

(*End definition for* `\l_tmpa_dim` *and* `\l_tmpb_dim`*. These variables are documented on page 82.*)

9.11 Creating and initialising skip variables

\skip_new:N
\skip_new:c

Allocation of a new internal registers.

```
⟨*package⟩
\cs_new_protected:Npn \skip_new:N #1
  {
    \__chk_if_free_cs:N #1
    \cs:w newskip \cs_end: #1
  }
⟨/package⟩
\cs_generate_variant:Nn \skip_new:N { c }
```

(*End definition for* `\skip_new:N` *and* `\skip_new:c`*. These functions are documented on page* **??**.)

\skip_const:Nn
\skip_const:cn

Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
\cs_new_protected:Npn \skip_const:Nn #1
  {
    \skip_new:N #1
    \skip_gset:Nn #1
  }
\cs_generate_variant:Nn \skip_const:Nn { c }
```

(*End definition for* `\skip_const:Nn` *and* `\skip_const:cn`*. These functions are documented on page* **??**.)

\skip_zero:N
\skip_zero:c
\skip_gzero:N
\skip_gzero:c

Reset the register to zero.

```
\cs_new_protected:Npn \skip_zero:N #1 { #1 \c_zero_skip }
\cs_new_protected:Npn \skip_gzero:N { \tex_global:D \skip_zero:N }
\cs_generate_variant:Nn \skip_zero:N  { c }
\cs_generate_variant:Nn \skip_gzero:N { c }
```

(*End definition for* `\skip_zero:N` *and* `\skip_zero:c`*. These functions are documented on page* **??**.)

\skip_zero_new:N
\skip_zero_new:c
\skip_gzero_new:N
\skip_gzero_new:c

Create a register if needed, otherwise clear it.

```
\cs_new_protected:Npn \skip_zero_new:N  #1
  { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
\cs_new_protected:Npn \skip_gzero_new:N #1
  { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
\cs_generate_variant:Nn \skip_zero_new:N  { c }
\cs_generate_variant:Nn \skip_gzero_new:N { c }
```

(*End definition for* `\skip_zero_new:N` *and others. These functions are documented on page* **??**.)

\skip_if_exist_p:N
\skip_if_exist_p:c
\skip_if_exist:NTF
\skip_if_exist:cTF

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \skip_if_exist:N \cs_if_exist:N { TF , T , F , p }
\prg_new_eq_conditional:NNn \skip_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(*End definition for* `\skip_if_exist:N` *and* `\skip_if_exist:c`*. These functions are documented on page* **??**.)

9.12 Setting skip variables

\skip_set:Nn
\skip_set:cn
\skip_gset:Nn
\skip_gset:cn

Much the same as for dimensions.

```
\cs_new_protected:Npn \skip_set:Nn #1#2
  { #1 ~ \etex_glueexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \skip_gset:Nn { \tex_global:D \skip_set:Nn }
\cs_generate_variant:Nn \skip_set:Nn  { c }
\cs_generate_variant:Nn \skip_gset:Nn { c }
```

(*End definition for* `\skip_set:Nn` *and* `\skip_set:cn`*. These functions are documented on page* **??**.)

\skip_set_eq:NN
\skip_set_eq:cN
\skip_set_eq:Nc
\skip_set_eq:cc
\skip_gset_eq:NN
\skip_gset_eq:cN
\skip_gset_eq:Nc
\skip_gset_eq:cc

All straightforward.

```
\cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
\cs_generate_variant:Nn \skip_set_eq:NN {       c }
\cs_generate_variant:Nn \skip_set_eq:NN { Nc , cc }
\cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
\cs_generate_variant:Nn \skip_gset_eq:NN {       c }
\cs_generate_variant:Nn \skip_gset_eq:NN { Nc , cc }
```

(*End definition for* `\skip_set_eq:NN` *and others. These functions are documented on page* **??**.)

\skip_add:Nn
\skip_add:cn
\skip_gadd:Nn
\skip_gadd:cn
\skip_sub:Nn
\skip_sub:cn
\skip_gsub:Nn
\skip_gsub:cn

Using `by` here deals with the (incorrect) case `\skip123`.

```
\cs_new_protected:Npn \skip_add:Nn #1#2
  { \tex_advance:D #1 by \etex_glueexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \skip_gadd:Nn { \tex_global:D \skip_add:Nn }
\cs_generate_variant:Nn \skip_add:Nn  { c }
\cs_generate_variant:Nn \skip_gadd:Nn { c }
\cs_new_protected:Npn \skip_sub:Nn #1#2
  { \tex_advance:D #1 by - \etex_glueexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \skip_gsub:Nn { \tex_global:D \skip_sub:Nn }
\cs_generate_variant:Nn \skip_sub:Nn  { c }
\cs_generate_variant:Nn \skip_gsub:Nn { c }
```

(*End definition for* `\skip_add:Nn` *and* `\skip_add:cn`*. These functions are documented on page* **??**.)

9.13 Skip expression conditionals

\skip_if_eq_p:nn
\skip_if_eq:nn*TF*

Comparing skips means doing two expansions to make strings, and then testing them. As a result, only equality is tested.

```
\prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
  {
    \if_int_compare:w
      \pdftex_strcmp:D { \skip_eval:n { #1 } } { \skip_eval:n { #2 } }
      = \c_zero
        \prg_return_true:
    \else:
        \prg_return_false:
    \fi:
  }
```

(*End definition for* `\skip_if_eq:nn`*. These functions are 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.

```
\cs_set_protected:Npn \__cs_tmp:w #1
  {
    \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
      {
        \exp_after:wN \__skip_if_finite:wwNw
        \skip_use:N \etex_glueexpr:D ##1 ; \prg_return_false:
        #1 ; \prg_return_true: \q_stop
      }
    \cs_new:Npn \__skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \q_stop {##3}
  }
\exp_args:No \__cs_tmp:w { \tl_to_str:n { fil } }
```

(*End definition for* `\skip_if_finite:n`. *These functions are documented on page 84.*)

9.14 Using skip expressions and variables

\skip_eval:n

Evaluating a skip expression expandably.

```
\cs_new:Npn \skip_eval:n #1
  { \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
\skip_use:c

Accessing a ⟨*skip*⟩.

```
\cs_new_eq:NN \skip_use:N \tex_the:D
\cs_generate_variant:Nn \skip_use:N { c }
```

(*End definition for* `\skip_use:N` *and* `\skip_use:c`. *These functions are documented on page* **??**.)

9.15 Inserting skips into the output

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

Inserting skips.

```
\cs_new_eq:NN  \skip_horizontal:N \tex_hskip:D
\cs_new:Npn \skip_horizontal:n #1
  { \skip_horizontal:N \etex_glueexpr:D #1 \scan_stop: }
\cs_new_eq:NN  \skip_vertical:N \tex_vskip:D
\cs_new:Npn \skip_vertical:n #1
  { \skip_vertical:N \etex_glueexpr:D #1 \scan_stop: }
\cs_generate_variant:Nn \skip_horizontal:N { c }
\cs_generate_variant:Nn \skip_vertical:N { c }
```

(*End definition for* `\skip_horizontal:N`, `\skip_horizontal:c`, *and* `\skip_horizontal:n`. *These functions are documented on page* **??**.)

9.16 Viewing skip variables

\skip_show:N
\skip_show:c

Diagnostics.

```
\cs_new_eq:NN  \skip_show:N \__kernel_register_show:N
\cs_generate_variant:Nn \skip_show:N { c }
```

(*End definition for* `\skip_show:N` *and* `\skip_show:c`*. These functions are documented on page* ??.)

\skip_show:n

Diagnostics. We don't use the TEX primitive `\showthe` to show skip expressions: this gives a more unified output, since the closing brace is read by the skip expression in all cases.

```
\cs_new_protected:Npn \skip_show:n #1
  { \etex_showtokens:D \exp_after:wN { \tex_the:D \etex_glueexpr:D #1 } }
```

(*End definition for* `\skip_show:n`*. This function is documented on page* 85.)

9.17 Constant skips

\c_zero_skip
\c_max_skip

Skips with no rubber component are just dimensions but need to terminate correctly.

```
\skip_const:Nn \c_zero_skip { \c_zero_dim }
\skip_const:Nn \c_max_skip { \c_max_dim }
```

(*End definition for* `\c_zero_skip` *and* `\c_max_skip`*. These functions are documented on page* 85.)

9.18 Scratch skips

\l_tmpa_skip
\l_tmpb_skip
\g_tmpa_skip
\g_tmpb_skip

We provide two local and two global scratch registers, maybe we need more or less.

```
\skip_new:N \l_tmpa_skip
\skip_new:N \l_tmpb_skip
\skip_new:N \g_tmpa_skip
\skip_new:N \g_tmpb_skip
```

(*End definition for* `\l_tmpa_skip` *and* `\l_tmpb_skip`*. These variables are documented on page* 85.)

9.19 Creating and initialising muskip variables

\muskip_new:N
\muskip_new:c

And then we add muskips.

```
⟨*package⟩
\cs_new_protected:Npn \muskip_new:N #1
  {
    \__chk_if_free_cs:N #1
    \cs:w newmuskip \cs_end: #1
  }
⟨/package⟩
\cs_generate_variant:Nn \muskip_new:N { c }
```

(*End definition for* `\muskip_new:N` *and* `\muskip_new:c`*. These functions are documented on page* ??.)

\muskip_const:Nn
\muskip_const:cn

Contrarily to integer constants, we cannot avoid using a register, even for constants.

```
\cs_new_protected:Npn \muskip_const:Nn #1
  {
    \muskip_new:N #1
    \muskip_gset:Nn #1
  }
\cs_generate_variant:Nn \muskip_const:Nn { c }
```

(*End definition for* `\muskip_const:Nn` *and* `\muskip_const:cn`*. These functions are documented on page* **??**.)

\muskip_zero:N
\muskip_zero:c
\muskip_gzero:N
\muskip_gzero:c

Reset the register to zero.

```
\cs_new_protected:Npn \muskip_zero:N #1
  { #1 \c_zero_muskip }
\cs_new_protected:Npn \muskip_gzero:N { \tex_global:D \muskip_zero:N }
\cs_generate_variant:Nn \muskip_zero:N  { c }
\cs_generate_variant:Nn \muskip_gzero:N { c }
```

(*End definition for* `\muskip_zero:N` *and* `\muskip_zero:c`*. These functions are documented on page* **??**.)

\muskip_zero_new:N
\muskip_zero_new:c
\muskip_gzero_new:N
\muskip_gzero_new:c

Create a register if needed, otherwise clear it.

```
\cs_new_protected:Npn \muskip_zero_new:N  #1
  { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
\cs_new_protected:Npn \muskip_gzero_new:N #1
  { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
\cs_generate_variant:Nn \muskip_zero_new:N  { c }
\cs_generate_variant:Nn \muskip_gzero_new:N { c }
```

(*End definition for* `\muskip_zero_new:N` *and others. These functions are documented on page* **??**.)

\muskip_if_exist_p:N
\muskip_if_exist_p:c
\muskip_if_exist:NTF
\muskip_if_exist:cTF

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \muskip_if_exist:N \cs_if_exist:N { TF , T , F , p }
\prg_new_eq_conditional:NNn \muskip_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(*End definition for* `\muskip_if_exist:N` *and* `\muskip_if_exist:c`*. These functions are documented on page* **??**.)

9.20 Setting muskip variables

\muskip_set:Nn
\muskip_set:cn
\muskip_gset:Nn
\muskip_gset:cn

This should be pretty familiar.

```
\cs_new_protected:Npn \muskip_set:Nn #1#2
  { #1 ~ \etex_muexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \muskip_gset:Nn { \tex_global:D \muskip_set:Nn }
\cs_generate_variant:Nn \muskip_set:Nn  { c }
\cs_generate_variant:Nn \muskip_gset:Nn { c }
```

(*End definition for* `\muskip_set:Nn` *and* `\muskip_set:cn`*. These functions are documented on page* **??**.)

\muskip_set_eq:NN
\muskip_set_eq:cN
\muskip_set_eq:Nc
\muskip_set_eq:cc
\muskip_gset_eq:NN
\muskip_gset_eq:cN
\muskip_gset_eq:Nc
\muskip_gset_eq:cc

All straightforward.

```
\cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
\cs_generate_variant:Nn \muskip_set_eq:NN {      c }
\cs_generate_variant:Nn \muskip_set_eq:NN { Nc , cc }
\cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
```

```
\cs_generate_variant:Nn \muskip_gset_eq:NN {       c }
\cs_generate_variant:Nn \muskip_gset_eq:NN { Nc , cc }
```

(*End definition for* `\muskip_set_eq:NN` *and others. These functions are documented on page* ??.)

`\muskip_add:Nn` `\muskip_add:cn` `\muskip_gadd:Nn` `\muskip_gadd:cn` `\muskip_sub:Nn` `\muskip_sub:cn` `\muskip_gsub:Nn` `\muskip_gsub:cn`

Using `by` here deals with the (incorrect) case `\muskip123`.

```
\cs_new_protected:Npn \muskip_add:Nn #1#2
  { \tex_advance:D #1 by \etex_muexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \muskip_gadd:Nn { \tex_global:D \muskip_add:Nn }
\cs_generate_variant:Nn \muskip_add:Nn  { c }
\cs_generate_variant:Nn \muskip_gadd:Nn { c }
\cs_new_protected:Npn \muskip_sub:Nn #1#2
  { \tex_advance:D #1 by - \etex_muexpr:D #2 \scan_stop: }
\cs_new_protected:Npn \muskip_gsub:Nn { \tex_global:D \muskip_sub:Nn }
\cs_generate_variant:Nn \muskip_sub:Nn  { c }
\cs_generate_variant:Nn \muskip_gsub:Nn { c }
```

(*End definition for* `\muskip_add:Nn` *and* `\muskip_add:cn`*. These functions are documented on page* ??.)

9.21 Using muskip expressions and variables

`\muskip_eval:n`

Evaluating a muskip expression expandably.

```
\cs_new:Npn \muskip_eval:n #1
  { \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*⟩.

```
\cs_new_eq:NN \muskip_use:N \tex_the:D
\cs_generate_variant:Nn \muskip_use:N { c }
```

(*End definition for* `\muskip_use:N` *and* `\muskip_use:c`*. These functions are documented on page* ??.)

9.22 Viewing muskip variables

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

Diagnostics.

```
\cs_new_eq:NN  \muskip_show:N \__kernel_register_show:N
\cs_generate_variant:Nn \muskip_show:N { c }
```

(*End definition for* `\muskip_show:N` *and* `\muskip_show:c`*. These functions are documented on page* ??.)

`\muskip_show:n`

Diagnostics. We don't use the TeX primitive `\showthe` to show muskip expressions: this gives a more unified output, since the closing brace is read by the muskip expression in all cases.

```
\cs_new_protected:Npn \muskip_show:n #1
  { \etex_showtokens:D \exp_after:wN { \tex_the:D \etex_muexpr:D #1 } }
```

(*End definition for* `\muskip_show:n`*. This function is documented on page 88.*)

9.23 Constant muskips

\c_zero_muskip
\c_max_muskip

Constant muskips given by their value.

```
\muskip_const:Nn \c_zero_muskip { 0 mu }
\muskip_const:Nn \c_max_muskip  { 16383.99999 mu }
```

(*End definition for* `\c_zero_muskip`*. This function is documented on page 88.*)

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.

```
\muskip_new:N \l_tmpa_muskip
\muskip_new:N \l_tmpb_muskip
\muskip_new:N \g_tmpa_muskip
\muskip_new:N \g_tmpb_muskip
```

(*End definition for* `\l_tmpa_muskip` *and* `\l_tmpb_muskip`*. These variables are documented on page 88.*)

9.25 Deprecated functions

\dim_case:nnn

Deprecated 2013-07-15.

```
\cs_new_eq:NN \dim_case:nnn \dim_case:nnF
```

(*End definition for* `\dim_case:nnn`*. This function is documented on page* **??**.)

```
⟨/initex | package⟩
```

10 l3tl implementation

```
⟨*initex | package⟩
```

```
⟨@@=tl⟩
```

A token list variable is a TeX macro that holds tokens. By using the ε-TeX primitive `\unexpanded` inside a TeX `\edef` it is possible to store any tokens, including `#`, in this way.

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.

```
\cs_new_protected:Npn \tl_new:N #1
  {
    \__chk_if_free_cs:N #1
    \cs_gset_eq:NN #1 \c_empty_tl
  }
\cs_generate_variant:Nn \tl_new:N { c }
```

(*End definition for* `\tl_new:N` *and* `\tl_new:c`*. These functions are documented on page* **??**.)

\tl_const:Nn
\tl_const:Nx
\tl_const:cn
\tl_const:cx

Constants are also easy to generate.

```
\cs_new_protected:Npn \tl_const:Nn #1#2
  {
    \__chk_if_free_cs:N #1
    \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
  }
\cs_new_protected:Npn \tl_const:Nx #1#2
  {
    \__chk_if_free_cs:N #1
    \cs_gset_nopar:Npx #1 {#2}
  }
\cs_generate_variant:Nn \tl_const:Nn { c }
\cs_generate_variant:Nn \tl_const:Nx { c }
```

(*End definition for* `\tl_const:Nn` *and others. These functions are documented on page* **??**.)

\tl_clear:N
\tl_clear:c
\tl_gclear:N
\tl_gclear:c

Clearing a token list variable means setting it to an empty value. Error checking will be sorted out by the parent function.

```
\cs_new_protected:Npn \tl_clear:N  #1
  { \tl_set_eq:NN #1 \c_empty_tl }
\cs_new_protected:Npn \tl_gclear:N #1
  { \tl_gset_eq:NN #1 \c_empty_tl }
\cs_generate_variant:Nn \tl_clear:N  { c }
\cs_generate_variant:Nn \tl_gclear:N { c }
```

(*End definition for* `\tl_clear:N` *and* `\tl_clear:c`*. These functions are documented on page* **??**.)

\tl_clear_new:N
\tl_clear_new:c
\tl_gclear_new:N
\tl_gclear_new:c

Clearing a token list variable means setting it to an empty value. Error checking will be sorted out by the parent function.

```
\cs_new_protected:Npn \tl_clear_new:N  #1
  { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
\cs_new_protected:Npn \tl_gclear_new:N #1
  { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
\cs_generate_variant:Nn \tl_clear_new:N  { c }
\cs_generate_variant:Nn \tl_gclear_new:N { c }
```

(*End definition for* `\tl_clear_new:N` *and* `\tl_clear_new:c`*. These functions are documented on page* **??**.)

\tl_set_eq:NN
\tl_set_eq:Nc
\tl_set_eq:cN
\tl_set_eq:cc
\tl_gset_eq:NN
\tl_gset_eq:Nc
\tl_gset_eq:cN
\tl_gset_eq:cc

For setting token list variables equal to each other.

```
\cs_new_eq:NN \tl_set_eq:NN  \cs_set_eq:NN
\cs_new_eq:NN \tl_set_eq:cN  \cs_set_eq:cN
\cs_new_eq:NN \tl_set_eq:Nc  \cs_set_eq:Nc
\cs_new_eq:NN \tl_set_eq:cc  \cs_set_eq:cc
\cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
\cs_new_eq:NN \tl_gset_eq:cN \cs_gset_eq:cN
\cs_new_eq:NN \tl_gset_eq:Nc \cs_gset_eq:Nc
\cs_new_eq:NN \tl_gset_eq:cc \cs_gset_eq:cc
```

(*End definition for* `\tl_set_eq:NN` *and others. These functions are documented on page* **??**.)

\tl_concat:NNN
\tl_concat:ccc
\tl_gconcat:NNN
\tl_gconcat:ccc

Concatenating token lists is easy.

```
\cs_new_protected:Npn \tl_concat:NNN #1#2#3
  { \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
\cs_new_protected:Npn \tl_gconcat:NNN #1#2#3
  { \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
\cs_generate_variant:Nn \tl_concat:NNN  { ccc }
\cs_generate_variant:Nn \tl_gconcat:NNN { ccc }
```

(*End definition for* `\tl_concat:NNN` *and* `\tl_concat:ccc`*. These functions are documented on page* ??.)

\tl_if_exist_p:N
\tl_if_exist_p:c
\tl_if_exist:NTF
\tl_if_exist:cTF

Copies of the cs functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }
\prg_new_eq_conditional:NNn \tl_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(*End definition for* `\tl_if_exist:N` *and* `\tl_if_exist:c`*. These functions are documented on page* ??.)

10.2 Constant token lists

\c_empty_tl

Never full. We need to define that constant before using `\tl_new:N`.

```
\tl_const:Nn \c_empty_tl { }
```

(*End definition for* `\c_empty_tl`*. This variable is documented on page 103.*)

\c_job_name_tl

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. LuaTeX does not quote file names containing spaces, whereas pdfTeX and XeTeX do. So there may be a correction to make in the LuaTeX case.

```
⟨*initex⟩
\luatex_if_engine:T
  {
    \tex_everyjob:D \exp_after:wN
      {
        \tex_the:D \tex_everyjob:D
        \lua_now_x:n
          { dofile ( assert ( kpse.find_file ("lualatexquotejobname.lua" ) ) ) }
      }
  }
\tex_everyjob:D \exp_after:wN
  {
    \tex_the:D \tex_everyjob:D
    \tl_const:Nx \c_job_name_tl { \tex_jobname:D }
  }
⟨/initex⟩
⟨*package⟩
\tl_const:Nx \c_job_name_tl { \tex_jobname:D }
⟨/package⟩
```

(*End definition for* `\c_job_name_tl`*. This variable is documented on page 103.*)

\c_space_tl

A space as a token list (as opposed to as a character).

```
\tl_const:Nn \c_space_tl { ~ }
```

(*End definition for* `\c_space_tl`*. This variable is documented on page 103.*)

10.3 Adding to token list variables

\tl_set:Nn
\tl_set:NV
\tl_set:Nv
\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

By using `\exp_not:n` token list variables can contain `#` tokens, which makes the token list registers provided by TEX more or less redundant. The `\tl_set:No` version is done "by hand" as it is used quite a lot.

```
\cs_new_protected:Npn \tl_set:Nn #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
\cs_new_protected:Npn \tl_set:No #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o {#2} } }
\cs_new_protected:Npn \tl_set:Nx #1#2
  { \cs_set_nopar:Npx #1 {#2} }
\cs_new_protected:Npn \tl_gset:Nn #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} } }
\cs_new_protected:Npn \tl_gset:No #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} } }
\cs_new_protected:Npn \tl_gset:Nx #1#2
  { \cs_gset_nopar:Npx #1 {#2} }
\cs_generate_variant:Nn \tl_set:Nn  {         NV , Nv , Nf }
\cs_generate_variant:Nn \tl_set:Nx  { c }
\cs_generate_variant:Nn \tl_set:Nn  { c, co , cV , cv , cf }
\cs_generate_variant:Nn \tl_gset:Nn {         NV , Nv , Nf }
\cs_generate_variant:Nn \tl_gset:Nx { c }
\cs_generate_variant:Nn \tl_gset:Nn { c, co , cV , cv , cf }
```

(*End definition for* `\tl_set:Nn` *and others. These functions are documented on page* ??.)

\tl_put_left:Nn
\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

Adding to the left is done directly to gain a little performance.

```
\cs_new_protected:Npn \tl_put_left:Nn #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_put_left:NV #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
\cs_new_protected:Npn \tl_put_left:No #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_put_left:Nx #1#2
  { \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 } }
\cs_new_protected:Npn \tl_gput_left:Nn #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_gput_left:NV #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
\cs_new_protected:Npn \tl_gput_left:No #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
\cs_new_protected:Npn \tl_gput_left:Nx #1#2
  { \cs_gset_nopar:Npx #1 { #2 \exp_not:o {#1} } }
\cs_generate_variant:Nn \tl_put_left:Nn  { c }
\cs_generate_variant:Nn \tl_put_left:NV  { c }
\cs_generate_variant:Nn \tl_put_left:No  { c }
\cs_generate_variant:Nn \tl_put_left:Nx  { c }
\cs_generate_variant:Nn \tl_gput_left:Nn { c }
\cs_generate_variant:Nn \tl_gput_left:NV { c }
\cs_generate_variant:Nn \tl_gput_left:No { c }
\cs_generate_variant:Nn \tl_gput_left:Nx { c }
```

(*End definition for* `\tl_put_left:Nn` *and others. These functions are documented on page* **??**.)

\tl_put_right:Nn
\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

The same on the right.

```
\cs_new_protected:Npn \tl_put_right:Nn #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\cs_new_protected:Npn \tl_put_right:NV #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\cs_new_protected:Npn \tl_put_right:No #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\cs_new_protected:Npn \tl_put_right:Nx #1#2
  { \cs_set_nopar:Npx #1 { \exp_not:o #1 #2 } }
\cs_new_protected:Npn \tl_gput_right:Nn #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\cs_new_protected:Npn \tl_gput_right:NV #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\cs_new_protected:Npn \tl_gput_right:No #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\cs_new_protected:Npn \tl_gput_right:Nx #1#2
  { \cs_gset_nopar:Npx #1 { \exp_not:o {#1} #2 } }
\cs_generate_variant:Nn \tl_put_right:Nn  { c }
\cs_generate_variant:Nn \tl_put_right:NV  { c }
\cs_generate_variant:Nn \tl_put_right:No  { c }
\cs_generate_variant:Nn \tl_put_right:Nx  { c }
\cs_generate_variant:Nn \tl_gput_right:Nn { c }
\cs_generate_variant:Nn \tl_gput_right:NV { c }
\cs_generate_variant:Nn \tl_gput_right:No { c }
\cs_generate_variant:Nn \tl_gput_right:Nx { c }
```

(*End definition for* `\tl_put_right:Nn` *and others. These functions are documented on page* **??**.)

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.

```
⟨*package⟩
\tex_ifodd:D \l@expl@check@declarations@bool
  \cs_set_protected:Npn \__cs_tmp:w #1
    {
      \if_meaning:w ? #1
        \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
      \fi:
      \use:x
        {
          \cs_set_protected:Npn #1 \exp_not:n { ##1 ##2 }
            {
              \__chk_if_exist_var:N \exp_not:n {##1}
              \exp_not:o { #1 {##1} {##2} }
            }
        }
      \__cs_tmp:w
```

```
    }
  \__cs_tmp:w
    \tl_set:Nn  \tl_set:No  \tl_set:Nx
    \tl_gset:Nn \tl_gset:No \tl_gset:Nx
    \tl_put_left:Nn   \tl_put_left:NV
    \tl_put_left:No   \tl_put_left:Nx
    \tl_gput_left:Nn  \tl_gput_left:NV
    \tl_gput_left:No  \tl_gput_left:Nx
    \tl_put_right:Nn  \tl_put_right:NV
    \tl_put_right:No  \tl_put_right:Nx
    \tl_gput_right:Nn \tl_gput_right:NV
    \tl_gput_right:No \tl_gput_right:Nx
    ? \q_recursion_stop
⟨/package⟩
```

The two `set_eq` functions are done by hand as the internals there are a bit different.

```
⟨*package⟩
  \cs_set_protected:Npn \tl_set_eq:NN #1#2
    {
      \__chk_if_exist_var:N #1
      \__chk_if_exist_var:N #2
      \cs_set_eq:NN #1 #2
    }
  \cs_set_protected:Npn \tl_gset_eq:NN #1#2
    {
      \__chk_if_exist_var:N #1
      \__chk_if_exist_var:N #2
      \cs_gset_eq:NN #1 #2
    }
⟨/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.

```
⟨*package⟩
  \cs_set_protected:Npn \tl_concat:NNN #1#2#3
    {
      \__chk_if_exist_var:N #1
      \__chk_if_exist_var:N #2
      \__chk_if_exist_var:N #3
      \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} }
    }
  \cs_set_protected:Npn \tl_gconcat:NNN #1#2#3
    {
      \__chk_if_exist_var:N #1
      \__chk_if_exist_var:N #2
      \__chk_if_exist_var:N #3
      \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} }
    }
\tex_fi:D
⟨/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 with two different category codes. This is set up here, while the detail is described below.

```
\group_begin:
  \tex_lccode:D `\A = `\@ \scan_stop:
  \tex_lccode:D `\B = `\@ \scan_stop:
  \tex_catcode:D `\A = 8 \scan_stop:
  \tex_catcode:D `\B = 3 \scan_stop:
\tex_lowercase:D
  {
    \group_end:
    \tl_const:Nn \c__tl_rescan_marker_tl { A B }
  }
```

(*End definition for* \c__tl_rescan_marker_tl*. This variable is documented on page* ??.)

\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_rescan:w

The idea here is to deal cleanly with the problem that \scantokens treats the argument as a file, and without the correct settings a TeX error occurs:

```
! File ended while scanning definition of ...
```

When expanding a token list this can be handled using \exp_not:N but this fails if the token list is not being expanded. So instead a delimited argument is used with an end marker which cannot appear within the token list which is scanned: two @ symbols with different category codes. The rescanned token list cannot contain the end marker, because all @ present in the token list are read with the same category code. As every character with charcode \newlinechar is replaced by the \endlinechar, and an extra \endlinechar is added at the end, we need to set both of those to -1, "unprintable".

```
\cs_new_protected_nopar:Npn \tl_set_rescan:Nnn
  { \__tl_set_rescan:NNnn \tl_set:Nn }
\cs_new_protected_nopar:Npn \tl_gset_rescan:Nnn
  { \__tl_set_rescan:NNnn \tl_gset:Nn }
\cs_new_protected_nopar:Npn \tl_rescan:nn
  { \__tl_set_rescan:NNnn \prg_do_nothing: \use:n }
\cs_new_protected:Npn \__tl_set_rescan:NNnn #1#2#3#4
  {
    \group_begin:
      \exp_args:No \etex_everyeof:D { \c__tl_rescan_marker_tl \exp_not:N }
      \tex_endlinechar:D \c_minus_one
      \tex_newlinechar:D \c_minus_one
      #3
      \use:x
        {
          \group_end:
          #1 \exp_not:N #2
            {
              \exp_after:wN \__tl_rescan:w
              \exp_after:wN \prg_do_nothing:
              \etex_scantokens:D {#4}
```

```
            }
        }
  }
\use:x
  {
    \cs_new:Npn \exp_not:N \__tl_rescan:w ##1
      \c__tl_rescan_marker_tl
      { \exp_not:N \exp_not:o { ##1 } }
  }
\cs_generate_variant:Nn \tl_set_rescan:Nnn  {     Nno , Nnx }
\cs_generate_variant:Nn \tl_set_rescan:Nnn  { c , cno , cnx }
\cs_generate_variant:Nn \tl_gset_rescan:Nnn {     Nno , Nnx }
\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 93.*)

10.5 Reassigning token list character codes

`\tl_to_lowercase:n`
`\tl_to_uppercase:n`

Just some names for a few primitives: we take care or wrapping the argument in braces.

```
\cs_new_protected:Npn \tl_to_lowercase:n #1
  { \tex_lowercase:D {#1} }
\cs_new_protected:Npn \tl_to_uppercase:n #1
  { \tex_uppercase:D {#1} }
```

(*End definition for* `\tl_to_lowercase:n`*. This function is documented on page 94.*)

10.6 Modifying token list variables

`\tl_replace_all:Nnn`
`\tl_replace_all:cnn`
`\tl_greplace_all:Nnn`
`\tl_greplace_all:cnn`
`\tl_replace_once:Nnn`
`\tl_replace_once:cnn`
`\tl_greplace_once:Nnn`
`\tl_greplace_once:cnn`
`\__tl_replace:NNNnn`
`\__tl_replace:w`
`\__tl_replace_all:`
`\__tl_replace_once:`
`\__tl_replace_once_end:w`

All of the replace functions are based on `\__tl_replace:NNNnn`, whose arguments are: ⟨*function*⟩, `\tl_(g)set:Nx`, ⟨*tl var*⟩, ⟨*search tokens*⟩, ⟨*replacement tokens*⟩.

```
\cs_new_protected_nopar:Npn \tl_replace_once:Nnn
  { \__tl_replace:NNNnn \__tl_replace_once: \tl_set:Nx }
\cs_new_protected_nopar:Npn \tl_greplace_once:Nnn
  { \__tl_replace:NNNnn \__tl_replace_once: \tl_gset:Nx }
\cs_new_protected_nopar:Npn \tl_replace_all:Nnn
  { \__tl_replace:NNNnn \__tl_replace_all: \tl_set:Nx }
\cs_new_protected_nopar:Npn \tl_greplace_all:Nnn
  { \__tl_replace:NNNnn \__tl_replace_all: \tl_gset:Nx }
\cs_generate_variant:Nn \tl_replace_once:Nnn  { c }
\cs_generate_variant:Nn \tl_greplace_once:Nnn { c }
\cs_generate_variant:Nn \tl_replace_all:Nnn   { c }
\cs_generate_variant:Nn \tl_greplace_all:Nnn  { c }
```

The idea is easier to understand by considering the case of `\tl_replace_all:Nnn`. The replacement happens within an `x`-type expansion. We use an auxiliary function `\__tl_tmp:w`, which essentially replaces the next ⟨*search tokens*⟩ by ⟨*replacement tokens*⟩. To avoid runaway arguments, we expand something like `\__tl_tmp:w` ⟨*token list*⟩ `\q_mark` ⟨*search tokens*⟩ `\q_stop`, repeating until the end. How do we detect that we have reached the last occurrence of ⟨*search tokens*⟩? The last replacement is characterized by the fact that the argument of `\__tl_tmp:w` contains `\q_mark`. In the code below, `\__tl_replace:w` takes an argument delimited by `\q_mark`, and removes the following token.

Before we reach the end, this gobbles `\q_mark \use_none_delimit_by_q_stop:w` which appear in the definition of `\__tl_tmp:w`, and leaves the ⟨*replacement tokens*⟩, passed to `\exp_not:n`, to be included in the `x`-expanding definition. At the end, the first `\q_mark` is within the argument of `\__tl_tmp:w`, and `\__tl_replace:w` gobbles the second `\q_mark` as well, leaving `\use_none_delimit_by_q_stop:w`, which ends the recursion cleanly.

```
\cs_new_protected:Npn \__tl_replace:NNNnn #1#2#3#4#5
  {
    \tl_if_empty:nTF {#4}
      {
        \__msg_kernel_error:nnx { kernel } { empty-search-pattern }
          { \tl_to_str:n {#5} }
      }
      {
        \group_align_safe_begin:
        \cs_set:Npx \__tl_tmp:w ##1##2 #4
          {
            ##2
            \exp_not:N \q_mark
            \exp_not:N \use_none_delimit_by_q_stop:w
            \exp_not:n { \exp_not:n {#5} }
            ##1
          }
        \group_align_safe_end:
        #2 #3
          {
            \exp_after:wN #1
            #3 \q_mark #4 \q_stop
          }
      }
  }
\cs_new:Npn \__tl_replace:w #1 \q_mark #2 { \exp_not:o {#1} }
```

The first argument of `\__tl_tmp:w` is responsible for repeating the replacement in the case of `replace_all`, and stopping it early for `replace_once`. Note also that we build `\__tl_tmp:w` within an `x`-expansion so that the ⟨*replacement tokens*⟩ can contain `#`. The second `\exp_not:n` ensures that the ⟨*replacement tokens*⟩ are not expanded by `\tl_(g)set:Nx`.

Now on to the difference between "once" and "all". The `\prg_do_nothing:` and accompanying `o`-expansion ensure that we don't lose braces in case the tokens between two occurrences of the ⟨*search tokens*⟩ form a brace group.

```
\cs_new_nopar:Npn \__tl_replace_all:
  {
    \exp_after:wN \__tl_replace:w
    \__tl_tmp:w \__tl_replace_all: \prg_do_nothing:
  }
\cs_new_nopar:Npn \__tl_replace_once:
  {
    \exp_after:wN \__tl_replace:w
```

```
    \__tl_tmp:w { \__tl_replace_once_end:w \prg_do_nothing: } \prg_do_nothing:
  }
\cs_new:Npn \__tl_replace_once_end:w #1 \q_mark #2 \q_stop
  { \exp_not:o {#1} }
```

(*End definition for* `\tl_replace_all:Nnn` *and* `\tl_replace_all:cnn`. *These functions are documented on page* **??**.)

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

Removal is just a special case of replacement.

```
\cs_new_protected:Npn \tl_remove_once:Nn #1#2
  { \tl_replace_once:Nnn #1 {#2} { } }
\cs_new_protected:Npn \tl_gremove_once:Nn #1#2
  { \tl_greplace_once:Nnn #1 {#2} { } }
\cs_generate_variant:Nn \tl_remove_once:Nn  { c }
\cs_generate_variant:Nn \tl_gremove_once:Nn { c }
```

(*End definition for* `\tl_remove_once:Nn` *and* `\tl_remove_once:cn`. *These functions are documented on page* **??**.)

`\tl_remove_all:Nn`
`\tl_remove_all:cn`
`\tl_gremove_all:Nn`
`\tl_gremove_all:cn`

Removal is just a special case of replacement.

```
\cs_new_protected:Npn \tl_remove_all:Nn #1#2
  { \tl_replace_all:Nnn #1 {#2} { } }
\cs_new_protected:Npn \tl_gremove_all:Nn #1#2
  { \tl_greplace_all:Nnn #1 {#2} { } }
\cs_generate_variant:Nn \tl_remove_all:Nn  { c }
\cs_generate_variant:Nn \tl_gremove_all:Nn { c }
```

10.7 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. For performance reasons, we hard-code the emptiness test done in `\tl_if_empty:n(TF)`: convert to harmless characters with `\tl_to_str:n`, and then use `\if_meaning:w \q_nil ... \q_nil`. Note that converting to a string is done after reading the delimited argument for `\use_none:n`. The similar construction `\exp_after:wN \use_none:n \tl_to_str:n {`⟨*token list*⟩`} ?` would fail if the token list contains the control sequence `\`, while `\escapechar` is a space or is unprintable.

```
\prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF }
  { \__tl_if_empty_return:o { \use_none:n #1 ? } }
\cs_generate_variant:Nn \tl_if_blank_p:n { V }
\cs_generate_variant:Nn \tl_if_blank:nT  { V }
\cs_generate_variant:Nn \tl_if_blank:nF  { V }
\cs_generate_variant:Nn \tl_if_blank:nTF { V }
\cs_generate_variant:Nn \tl_if_blank_p:n { o }
\cs_generate_variant:Nn \tl_if_blank:nT  { o }
\cs_generate_variant:Nn \tl_if_blank:nF  { o }
\cs_generate_variant:Nn \tl_if_blank:nTF { o }
```

(*End definition for* `\tl_remove_all:Nn` *and* `\tl_remove_all:cn`. *These functions are documented on page* **??**.)

\tl_if_empty_p:N
\tl_if_empty_p:c
\tl_if_empty:NTF
\tl_if_empty:cTF

These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).

```
\prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \c_empty_tl
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \tl_if_empty_p:N { c }
\cs_generate_variant:Nn \tl_if_empty:NT  { c }
\cs_generate_variant:Nn \tl_if_empty:NF  { c }
\cs_generate_variant:Nn \tl_if_empty:NTF { c }
```

(*End definition for* `\tl_if_empty:N` *and* `\tl_if_empty:c`*. These functions are documented on page* **??**.)

\tl_if_empty_p:n
\tl_if_empty_p:V
\tl_if_empty:nTF
\tl_if_empty:VTF

It would be tempting to just use `\if_meaning:w \q_nil #1 \q_nil` as a test since this works really well. However, it 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… A safer route is to convert the entire token list into harmless characters first and then compare that. This way the test will even accept `\q_nil` as the first token.

```
\prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
  {
    \exp_after:wN \if_meaning:w \exp_after:wN \q_nil \tl_to_str:n {#1} \q_nil
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \tl_if_empty_p:n { V }
\cs_generate_variant:Nn \tl_if_empty:nTF { V }
\cs_generate_variant:Nn \tl_if_empty:nT  { V }
\cs_generate_variant:Nn \tl_if_empty:nF  { V }
```

(*End definition for* `\tl_if_empty:n` *and* `\tl_if_empty:V`*. These functions are documented on page* **??**.)

\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 conditionals on token lists, which mostly 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:n(TF)`, but the expansion is hard-coded for efficiency, as this auxiliary function is used in many places. Note that this works because `\tl_to_str:n` expands tokens that follow until reading a catcode 1 (begin-group) token.

```
\cs_new:Npn \__tl_if_empty_return:o #1
  {
    \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
      \tl_to_str:n \exp_after:wN {#1} \q_nil
```

```
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F }
  { \__tl_if_empty_return:o {#1} }
```

(*End definition for* `\tl_if_empty:o`. *These functions are documented on page* **??**.)

`\tl_if_eq_p:NN`
`\tl_if_eq_p:Nc`
`\tl_if_eq_p:cN`
`\tl_if_eq_p:cc`
`\tl_if_eq:NNTF`
`\tl_if_eq:NcTF`
`\tl_if_eq:cNTF`
`\tl_if_eq:ccTF`

Returns `\c_true_bool` if and only if the two token list variables are equal.

```
\prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF }
  {
    \if_meaning:w #1 #2
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \tl_if_eq_p:NN { Nc , c , cc }
\cs_generate_variant:Nn \tl_if_eq:NNTF { Nc , c , cc }
\cs_generate_variant:Nn \tl_if_eq:NNT  { Nc , c , cc }
\cs_generate_variant:Nn \tl_if_eq:NNF  { Nc , c , cc }
```

(*End definition for* `\tl_if_eq:NN` *and others. These functions are documented on page* **??**.)

`\tl_if_eq:nnTF`
`\l__tl_internal_a_tl`
`\l__tl_internal_b_tl`

A simple store and compare routine.

```
\prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F ,  TF }
  {
    \group_begin:
      \tl_set:Nn \l__tl_internal_a_tl {#1}
      \tl_set:Nn \l__tl_internal_b_tl {#2}
      \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
        \group_end:
        \prg_return_true:
      \else:
        \group_end:
        \prg_return_false:
      \fi:
  }
\tl_new:N \l__tl_internal_a_tl
\tl_new:N \l__tl_internal_b_tl
```

(*End definition for* `\tl_if_eq:nnTF`. *This function is documented on page* **??**.)

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

See `\tl_if_in:nn(TF)` for further comments. Here we simply expand the token list variable and pass it to `\tl_if_in:nn(TF)`.

```
\cs_new_protected_nopar:Npn \tl_if_in:NnT  { \exp_args:No \tl_if_in:nnT  }
\cs_new_protected_nopar:Npn \tl_if_in:NnF  { \exp_args:No \tl_if_in:nnF  }
\cs_new_protected_nopar:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
\cs_generate_variant:Nn \tl_if_in:NnT { c }
\cs_generate_variant:Nn \tl_if_in:NnF  { c }
\cs_generate_variant:Nn \tl_if_in:NnTF { c }
```

(*End definition for* `\tl_if_in:NnTF` *and* `\tl_if_in:cnTF`. *These functions are documented on page* **??**.)

`\tl_if_in:nnTF`
`\tl_if_in:VnTF`
`\tl_if_in:onTF`
`\tl_if_in:noTF`

Once more, the test relies on `\tl_to_str:n` 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:n(TF)` for details on the emptiness test.

Special care is needed to treat correctly cases like `\tl_if_in:nnTF {a state}{states}`, where `#1#2` contains `#2` before the end. 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.

```
\prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T  , F , TF }
  {
    \cs_set:Npn \__tl_tmp:w ##1 #2 { }
    \tl_if_empty:oTF { \__tl_tmp:w #1 {} {} #2 }
      { \prg_return_false: } { \prg_return_true: }
  }
\cs_generate_variant:Nn \tl_if_in:nnT  { V , o , no }
\cs_generate_variant:Nn \tl_if_in:nnF  { V , o , no }
\cs_generate_variant:Nn \tl_if_in:nnTF { V , o , no }
```

(*End definition for* `\tl_if_in:nnTF` *and others. These functions are documented on page* **??**.)

`\tl_if_single_p:N`
`\tl_if_single:NTF`

Expand the token list and feed it to `\tl_if_single:n`.

```
\cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
\cs_new:Npn \tl_if_single:NT  { \exp_args:No \tl_if_single:nT  }
\cs_new:Npn \tl_if_single:NF  { \exp_args:No \tl_if_single:nF  }
\cs_new:Npn \tl_if_single:NTF { \exp_args:No \tl_if_single:nTF }
```

(*End definition for* `\tl_if_single:N`. *These functions are documented on page 95*.)

`\tl_if_single_p:n`
`\tl_if_single:nTF`

A token list has exactly one item if it is a single token or a single brace group, surrounded by optional explicit spaces. The naive version of this test would do `\use_none:n #1`, and test if the result is empty. However, this will fail when the token list is empty. Furthermore, it does not allow optional trailing spaces.

```
\prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
  { \__str_if_eq_x_return:nn { \exp_not:o { \use_none:nn #1 ?? } } {?} }
```

(*End definition for* `\tl_if_single:n`. *These functions are documented on page 95*.)

`\tl_case:Nn`
`\tl_case:cn`
`\tl_case:NnTF`
`\tl_case:cnTF`
`\__tl_case:nnTF`
`\__tl_case:Nw`
`\__tl_case_end:nw`

Similar set up to `\str_case:nn(TF)` as described in l3basics.

```
\cs_new:Npn \tl_case:Nn #1#2
  {
    \tex_romannumeral:D
    \__tl_case:NnTF #1 {#2} { } { }
  }
\cs_new:Npn \tl_case:NnT #1#2#3
  {
    \tex_romannumeral:D
    \__tl_case:NnTF #1 {#2} {#3} { }
  }
```

```
\cs_new:Npn \tl_case:NnF #1#2#3
  {
    \tex_romannumeral:D
    \__tl_case:NnTF #1 {#2} { } {#3}
  }
\cs_new:Npn \tl_case:NnTF #1#2
  {
    \tex_romannumeral:D
    \__tl_case:NnTF #1 {#2}
  }
\cs_new:Npn \__tl_case:NnTF #1#2#3#4
  { \__tl_case:Nw #1 #2 #1 { } \q_mark {#3} \q_mark {#4} \q_stop }
\cs_new:Npn \__tl_case:Nw #1#2#3
  {
    \tl_if_eq:NNTF #1 #2
      { \__tl_case_end:nw {#3} }
      { \__tl_case:Nw #1 }
  }
\cs_generate_variant:Nn \tl_case:Nn   { c }
\cs_generate_variant:Nn \tl_case:NnT  { c }
\cs_generate_variant:Nn \tl_case:NnF  { c }
\cs_generate_variant:Nn \tl_case:NnTF { c }
\cs_new_eq:NN \__tl_case_end:nw \__prg_case_end:nw
```

(*End definition for* `\tl_case:Nn` *and* `\tl_case:cn`*. These functions are documented on page* **??**.)

10.8 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.

```
\cs_new:Npn \tl_map_function:nN #1#2
  {
    \__tl_map_function:Nn #2 #1
      \q_recursion_tail
    \__prg_break_point:Nn \tl_map_break: { }
  }
\cs_new_nopar:Npn \tl_map_function:NN
  { \exp_args:No \tl_map_function:nN }
\cs_new:Npn \__tl_map_function:Nn #1#2
  {
    \__quark_if_recursion_tail_break:nN {#2} \tl_map_break:
    #1 {#2} \__tl_map_function:Nn #1
  }
\cs_generate_variant:Nn \tl_map_function:NN { c }
```

(*End definition for* `\tl_map_function:nN`*. This function is documented on page* **??**.)

\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.

```
\cs_new_protected:Npn \tl_map_inline:nn #1#2
  {
    \int_gincr:N \g__prg_map_int
    \cs_gset:cpn { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}
    \exp_args:Nc \__tl_map_function:Nn
      { __prg_map_ \int_use:N \g__prg_map_int :w }
      #1 \q_recursion_tail
    \__prg_break_point:Nn \tl_map_break: { \int_gdecr:N \g__prg_map_int }
  }
\cs_new_protected:Npn \tl_map_inline:Nn
  { \exp_args:No \tl_map_inline:nn }
\cs_generate_variant:Nn \tl_map_inline:Nn { c }
```

(*End definition for* `\tl_map_inline:nn`. *This function is documented on page* ??.)

`\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*⟩.

```
\cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
  {
    \__tl_map_variable:Nnn #2 {#3} #1
      \q_recursion_tail
    \__prg_break_point:Nn \tl_map_break: { }
  }
\cs_new_protected_nopar:Npn \tl_map_variable:NNn
  { \exp_args:No \tl_map_variable:nNn }
\cs_new_protected:Npn \__tl_map_variable:Nnn #1#2#3
  {
    \tl_set:Nn #1 {#3}
    \__quark_if_recursion_tail_break:NN #1 \tl_map_break:
    \use:n {#2}
    \__tl_map_variable:Nnn #1 {#2}
  }
\cs_generate_variant:Nn \tl_map_variable:NNn { c }
```

(*End definition for* `\tl_map_variable:nNn`. *This function is documented on page* ??.)

`\tl_map_break:`
`\tl_map_break:n`

The break statements use the general `\__prg_map_break:Nn`.

```
\cs_new_nopar:Npn \tl_map_break:
  { \__prg_map_break:Nn \tl_map_break: { } }
\cs_new_nopar:Npn \tl_map_break:n
  { \__prg_map_break:Nn \tl_map_break: }
```

(*End definition for* `\tl_map_break:`. *This function is documented on page* 97.)

10.9 Using token lists

`\tl_to_str:n`

Another name for a primitive.

```
\cs_new_eq:NN \tl_to_str:n \etex_detokenize:D
```

(*End definition for* `\tl_to_str:n`. *This function is documented on page* 98.)

\tl_to_str:N
\tl_to_str:c

These functions return the replacement text of a token list as a string.

```
\cs_new:Npn \tl_to_str:N #1 { \etex_detokenize:D \exp_after:wN {#1} }
\cs_generate_variant:Nn \tl_to_str:N { c }
```

(*End definition for* `\tl_to_str:N` *and* `\tl_to_str:c`*. These functions are documented on page* **??**.)

\tl_use:N
\tl_use:c

Token lists which are simply not defined will give a clear TEX error here. No such luck for ones equal to `\scan_stop:` so instead a test is made and if there is an issue an error is forced.

```
\cs_new:Npn \tl_use:N #1
  {
    \tl_if_exist:NTF #1 {#1}
      { \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#1} }
  }
\cs_generate_variant:Nn \tl_use:N { c }
```

(*End definition for* `\tl_use:N` *and* `\tl_use:c`*. These functions are documented on page* **??**.)

10.10 Working with the contents of token lists

\tl_count:n
\tl_count:V
\tl_count:o
\tl_count:N
\tl_count:c
__tl_count:n

Count number of elements within a token list or token list variable. Brace groups within the list are read as a single element. Spaces are ignored. `\__tl_count:n` grabs the element and replaces it by `+1`. The `0` to ensure it works on an empty list.

```
\cs_new:Npn \tl_count:n #1
  {
    \int_eval:n
      { 0 \tl_map_function:nN {#1} \__tl_count:n }
  }
\cs_new:Npn \tl_count:N #1
  {
    \int_eval:n
      { 0 \tl_map_function:NN #1 \__tl_count:n }
  }
\cs_new:Npn \__tl_count:n #1 { + \c_one }
\cs_generate_variant:Nn \tl_count:n { V , o }
\cs_generate_variant:Nn \tl_count:N { c }
```

(*End definition for* `\tl_count:n`*,* `\tl_count:V`*, and* `\tl_count:o`*. These functions are documented on page* **??**.)

\tl_reverse_items:n
__tl_reverse_items:nwNwn
__tl_reverse_items:wn

Reversal of a token list is done by taking one item at a time and putting it after `\q_stop`.

```
\cs_new:Npn \tl_reverse_items:n #1
  {
    \__tl_reverse_items:nwNwn #1 ?
      \q_mark \__tl_reverse_items:nwNwn
      \q_mark \__tl_reverse_items:wn
      \q_stop { }
  }
\cs_new:Npn \__tl_reverse_items:nwNwn #1 #2 \q_mark #3 #4 \q_stop #5
  {
    #3 #2
```

```
    \q_mark \__tl_reverse_items:nwNwn
    \q_mark \__tl_reverse_items:wn
    \q_stop { {#1} #5 }
  }
\cs_new:Npn \__tl_reverse_items:wn #1 \q_stop #2
  { \exp_not:o { \use_none:nn #2 } }
```

(*End definition for* `\tl_reverse_items:n`*. This function is documented on page* 99*.*)

`\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.

```
\cs_new:Npn \tl_trim_spaces:n #1
  { \__tl_trim_spaces:nn { \q_mark #1 } \exp_not:o }
\cs_new_protected:Npn \tl_trim_spaces:N #1
  { \tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
\cs_new_protected:Npn \tl_gtrim_spaces:N #1
  { \tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
\cs_generate_variant:Nn \tl_trim_spaces:N  { c }
\cs_generate_variant:Nn \tl_gtrim_spaces:N { c }
```

(*End definition for* `\tl_trim_spaces:n`*. This function is documented on page* ??*.*)

`\__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*⟩.

```
\cs_set:Npn \__tl_tmp:w #1
  {
    \cs_new:Npn \__tl_trim_spaces:nn ##1
      {
        \__tl_trim_spaces_auxi:w
          ##1
          \q_nil
          \q_mark #1 { }
          \q_mark \__tl_trim_spaces_auxii:w
          \__tl_trim_spaces_auxiii:w
          #1 \q_nil
          \__tl_trim_spaces_auxiv:w
        \q_stop
      }
    \cs_new:Npn \__tl_trim_spaces_auxi:w ##1 \q_mark #1 ##2 \q_mark ##3
```

```
      {
        ##3
        \__tl_trim_spaces_auxi:w
        \q_mark
        ##2
        \q_mark #1 {##1}
      }
    \cs_new:Npn \__tl_trim_spaces_auxii:w
        \__tl_trim_spaces_auxi:w \q_mark \q_mark ##1
      {
        \__tl_trim_spaces_auxiii:w
        ##1
      }
    \cs_new:Npn \__tl_trim_spaces_auxiii:w ##1 #1 \q_nil ##2
      {
        ##2
        ##1 \q_nil
        \__tl_trim_spaces_auxiii:w
      }
    \cs_new:Npn \__tl_trim_spaces_auxiv:w ##1 \q_nil ##2 \q_stop ##3
      { ##3 { \use_none:n ##1 } }
  }
\__tl_tmp:w { ~ }
```

(*End definition for* `\__tl_trim_spaces:nn`.)

10.11 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`. *These variables are documented on page* ??.)

`\__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 `\romannumeral` and ends by producing `\c_zero` 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`.

```
\cs_new:Npn \__tl_act:NNNnn #1#2#3#4#5
  {
    \group_align_safe_begin:
    \__tl_act_loop:w #5 \q___tl_act_mark \q___tl_act_stop
    {#4} #1 #2 #3
    \__tl_act_result:n { }
  }
```

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 `\c_zero` and the result in the input stream, to terminate the expansion of `\romannumeral`. 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.

```
\cs_new:Npn \__tl_act_loop:w #1 \q___tl_act_stop
  {
    \tl_if_head_is_N_type:nTF {#1}
      { \__tl_act_normal:NwnNNN }
      {
        \tl_if_head_is_group:nTF {#1}
          { \__tl_act_group:nwnNNN }
          { \__tl_act_space:wwnNNN }
      }
    #1 \q___tl_act_stop
  }
\cs_new:Npn \__tl_act_normal:NwnNNN #1 #2 \q___tl_act_stop #3#4
  {
    \if_meaning:w \q___tl_act_mark #1
      \exp_after:wN \__tl_act_end:wn
    \fi:
    #4 {#3} #1
    \__tl_act_loop:w #2 \q___tl_act_stop
    {#3} #4
  }
\cs_new:Npn \__tl_act_end:wn #1 \__tl_act_result:n #2
  { \group_align_safe_end: \c_zero #2 }
\cs_new:Npn \__tl_act_group:nwnNNN #1 #2 \q___tl_act_stop #3#4#5
  {
    #5 {#3} {#1}
    \__tl_act_loop:w #2 \q___tl_act_stop
    {#3} #4 #5
  }
\exp_last_unbraced:NNo
  \cs_new:Npn \__tl_act_space:wwnNNN \c_space_tl #1 \q___tl_act_stop #2#3#4#5
  {
    #5 {#2}
    \__tl_act_loop:w #1 \q___tl_act_stop
    {#2} #3 #4 #5
  }
```

Typically, the output is done to the right of what was already output, using `\__tl_act_output:n`, but for the `\__tl_act_reverse` functions, it should be done to the left.

```
\cs_new:Npn \__tl_act_output:n #1 #2 \__tl_act_result:n #3
  { #2 \__tl_act_result:n { #3 #1 } }
\cs_new:Npn \__tl_act_reverse_output:n #1 #2 \__tl_act_result:n #3
  { #2 \__tl_act_result:n { #1 #3 } }
```

(*End definition for `\__tl_act:NNNnn`.*)

\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.

```
\cs_new:Npn \tl_reverse:n #1
  {
    \etex_unexpanded:D \exp_after:wN
      {
        \tex_romannumeral:D
        \__tl_act:NNNnn
          \__tl_reverse_normal:nN
          \__tl_reverse_group_preserve:nn
          \__tl_reverse_space:n
          { }
          {#1}
      }
  }
\cs_generate_variant:Nn \tl_reverse:n { o , V }
\cs_new:Npn \__tl_reverse_normal:nN #1#2
  { \__tl_act_reverse_output:n {#2} }
\cs_new:Npn \__tl_reverse_group_preserve:nn #1#2
  { \__tl_act_reverse_output:n { {#2} } }
\cs_new:Npn \__tl_reverse_space:n #1
  { \__tl_act_reverse_output:n { ~ } }
```

(*End definition for* `\tl_reverse:n`*,* `\tl_reverse:o`*, and* `\tl_reverse:V`*. These functions are documented on page* **??**.)

\tl_reverse:N
\tl_reverse:c
\tl_greverse:N
\tl_greverse:c

This reverses the list, leaving `\exp_stop_f:` in front, which stops the `f`-expansion.

```
\cs_new_protected:Npn \tl_reverse:N #1
  { \tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
\cs_new_protected:Npn \tl_greverse:N #1
  { \tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
\cs_generate_variant:Nn \tl_reverse:N  { c }
\cs_generate_variant:Nn \tl_greverse:N { c }
```

(*End definition for* `\tl_reverse:N` *and others. These functions are documented on page* **??**.)

10.12 The first token from a token list

\tl_head:N
\tl_head:n
\tl_head:V
\tl_head:v
\tl_head:f
__tl_head_auxi:nw
__tl_head_auxii:nw
\tl_head:w
\tl_tail:N
\tl_tail:n
\tl_tail:V
\tl_tail:v
\tl_tail:f

Finding the head of a token list expandably 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.

```
\cs_new:Npn \tl_head:n #1
  {
    \etex_unexpanded:D
      \if_false: { \fi: \__tl_head_auxi:nw #1 { } \q_stop }
  }
\cs_new:Npn \__tl_head_auxi:nw #1#2 \q_stop
  { \exp_after:wN \__tl_head_auxii:nw \exp_after:wN { \if_false: } \fi: {#1} }
\cs_new:Npn \__tl_head_auxii:nw #1
  {
    \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
      \tl_to_str:n \exp_after:wN { \use_none:n #1 } \q_nil
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
      {#1}
      { \if_false: { \fi: \__tl_head_auxi:nw #1 } }
  }
\cs_generate_variant:Nn \tl_head:n { V , v , f }
\cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
\cs_new_nopar: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 `\etex_unexpanded:D`. While we could optimise the test here, this would leave some tokens "banned" in the input, which we do not have with this definition.

```
\cs_new:Npn \tl_tail:n #1
  {
    \etex_unexpanded:D
      \tl_if_blank:nTF {#1}
        { { } }
        { \exp_after:wN { \use_none:n #1 } }
  }
\cs_generate_variant:Nn \tl_tail:n { V , v , f }
\cs_new_nopar:Npn \tl_tail:N { \exp_args:No \tl_tail:n }
```

(*End definition for* `\tl_head:N` *and others. These functions are documented on page* ??.)

`\str_head:n` `\str_tail:n` `\__str_head:w` `\__str_tail:w`

After `\tl_to_str:n`, we have a list of character tokens, all with category code 12, except the space, which has category code 10. Directly using `\tl_head:w` would thus lose leading spaces. Instead, we take an argument delimited by an explicit space, and then only use `\tl_head:w`. If the string started with a space, then the argument of `\__str_head:w` is empty, and the function correctly returns a space character. Otherwise, it returns the first token of `#1`, which is the first token of the string. If the string is empty, we return an empty result.

To remove the first character of `\tl_to_str:n {#1}`, we test it using `\if_charcode:w \scan_stop:`, always `false` for characters. If the argument was non-empty, then `\__str_tail:w` returns everything until the first `X` (with category code letter, no risk of confusing with the user input). If the argument was empty, the first `X` is taken by `\if_charcode:w`, and nothing is returned. We use `X` as a ⟨*marker*⟩, rather than a quark because the test `\if_charcode:w \scan_stop:` ⟨*marker*⟩ has to be `false`.

```
\cs_new:Npn \str_head:n #1
  {
    \exp_after:wN \__str_head:w
    \tl_to_str:n {#1}
    { { } } ~ \q_stop
  }
\cs_new:Npn \__str_head:w #1 ~ %
  { \tl_head:w #1 { ~ } }
\cs_new:Npn \str_tail:n #1
  {
    \exp_after:wN \__str_tail:w
    \reverse_if:N \if_charcode:w
        \scan_stop: \tl_to_str:n {#1} X X \q_stop
  }
\cs_new:Npn \__str_tail:w #1 X #2 \q_stop { \fi: #1 }
```

(*End definition for* `\str_head:n` *and* `\str_tail:n`*. These functions are documented on page* *101*.)

`\tl_if_head_eq_meaning_p:nN`
`\tl_if_head_eq_meaning:nNTF`
`\tl_if_head_eq_charcode_p:nN`
`\tl_if_head_eq_charcode:nNTF`
`\tl_if_head_eq_charcode_p:fN`
`\tl_if_head_eq_charcode:fNTF`
`\tl_if_head_eq_catcode_p:nN`
`\tl_if_head_eq_catcode:nNTF`

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`.

```
\prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
  {
```

```
    \if_charcode:w
        \exp_not:N #2
        \tl_if_head_is_N_type:nTF { #1 ? }
          {
            \exp_after:wN \exp_not:N
            \tl_head:w #1 { ? \use_none:nn } \q_stop
          }
          { \str_head:n {#1} }
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \tl_if_head_eq_charcode_p:nN { f }
\cs_generate_variant:Nn \tl_if_head_eq_charcode:nNTF { f }
\cs_generate_variant:Nn \tl_if_head_eq_charcode:nNT  { f }
\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`.

```
\prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
  {
    \if_catcode:w
        \exp_not:N #2
        \tl_if_head_is_N_type:nTF { #1 ? }
          {
            \exp_after:wN \exp_not:N
            \tl_head:w #1 { ? \use_none:nn } \q_stop
          }
          {
            \tl_if_head_is_group:nTF {#1}
              { \c_group_begin_token }
              { \c_space_token }
          }
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

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.

```
\prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
  {
    \tl_if_head_is_N_type:nTF { #1 ? }
      { \__tl_if_head_eq_meaning_normal:nN }
      { \__tl_if_head_eq_meaning_special:nN }
    {#1} #2
  }
\cs_new:Npn \__tl_if_head_eq_meaning_normal:nN #1 #2
  {
    \exp_after:wN \if_meaning:w
        \tl_head:w #1 { ?? \use_none:nnn } \q_stop #2
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \__tl_if_head_eq_meaning_special:nN #1 #2
  {
    \if_charcode:w \str_head:n {#1} \exp_not:N #2
      \exp_after:wN \use:n
    \else:
      \prg_return_false:
      \exp_after:wN \use_none:n
    \fi:
    {
      \if_catcode:w \exp_not:N #2
                    \tl_if_head_is_group:nTF {#1}
                      { \c_group_begin_token }
                      { \c_space_token }
        \prg_return_true:
      \else:
        \prg_return_false:
      \fi:
    }
  }
```

(*End definition for* `\tl_if_head_eq_meaning:nN`. *These functions are documented on page 101.*)

`\tl_if_head_is_N_type_p:n`
`\tl_if_head_is_N_type:nTF`

The first token of a token list can be either an N-type argument, a begin-group token (catcode 1), or an explicit space token (catcode 10 and charcode 32). The latter two cases are characterized by the fact that `\use:n` removes some tokens from `#1`, hence changing its string representation (no token can have an empty string representation). The extra brace group covers the case of an empty argument, whose head is not “normal”.

```
\prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
  {
    \__str_if_eq_x_return:nn
      { \exp_not:o { \use:n #1 { } } }
      { \exp_not:n { #1 { } } }
```

```
  }
```

(*End definition for* `\tl_if_head_is_N_type:n`*. These functions are documented on page 102.*)

`\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.[5]

```
\prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
  {
    \if_catcode:w *
        \exp_after:wN \use_none:n
          \exp_after:wN {
            \exp_after:wN {
              \token_to_str:N #1 ?
            }
          }
          *
      \prg_return_false:
    \else:
      \prg_return_true:
    \fi:
  }
```

(*End definition for* `\tl_if_head_is_group:n`*. These functions are documented on page 102.*)

`\tl_if_head_is_space_p:n`
`\tl_if_head_is_space:nTF`
`\__tl_if_head_is_space:w`

If the first token of the token list is an explicit space, i.e., a character token with character code 32 and category code 10, then this test will be `true`. It is `false` if the token list is empty, if the first token is an implicit space token, such as `\c_space_token`, or any token other than an explicit space. The slightly convoluted approach with `\romannumeral` ensures that each expansion step gives a balanced token list.

```
\prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
  {
    \tex_romannumeral:D \if_false: { \fi:
      \__tl_if_head_is_space:w ? #1 ? ~ }
  }
\cs_new:Npn \__tl_if_head_is_space:w #1 ~
  {
    \tl_if_empty:oTF { \use_none:n #1 }
      { \exp_after:wN \c_zero \exp_after:wN \prg_return_true: }
      { \exp_after:wN \c_zero \exp_after:wN \prg_return_false: }
    \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
  }
```

(*End definition for* `\tl_if_head_is_space:n`*. These functions are documented on page 102.*)

[5] 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.

10.13 Viewing token lists

\tl_show:N
\tl_show:c

Showing token list variables is done after checking that the variable is defined (see __kernel_register_show:N.

```
\cs_new_protected:Npn \tl_show:N #1
  {
    \tl_if_exist:NTF #1
      { \cs_show:N #1 }
      {
        \__msg_kernel_error:nnx { kernel } { variable-not-defined }
          { \token_to_str:N #1 }
      }
  }
\cs_generate_variant:Nn \tl_show:N { c }
```

(*End definition for* \tl_show:N *and* \tl_show:c*. These functions are documented on page* **??**.)

\tl_show:n

The __msg_show_variable:n internal function performs line-wrapping, removes a leading >␣, then 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.

```
\cs_new_protected:Npn \tl_show:n #1
  { \__msg_show_variable:n { > ~ \tl_to_str:n {#1} } }
```

(*End definition for* \tl_show:n*. This function is documented on page 102.*)

10.14 Scratch token lists

\g_tmpa_tl
\g_tmpb_tl

Global temporary token list variables. They are supposed to be set and used immediately, with no delay between the definition and the use because you can't count on other macros not to redefine them from under you.

```
\tl_new:N \g_tmpa_tl
\tl_new:N \g_tmpb_tl
```

(*End definition for* \g_tmpa_tl *and* \g_tmpb_tl*. These variables are documented on page 103.*)

\l_tmpa_tl
\l_tmpb_tl

These are local temporary token list variables. Be sure not to assume that the value you put into them will survive for long—see discussion above.

```
\tl_new:N \l_tmpa_tl
\tl_new:N \l_tmpb_tl
```

(*End definition for* \l_tmpa_tl *and* \l_tmpb_tl*. These variables are documented on page 103.*)

10.15 Deprecated functions

\tl_case:Nnn
\tl_case:cnn

Deprecated 2013-07-15.

```
\cs_new_eq:NN \tl_case:Nnn \tl_case:NnF
\cs_new_eq:NN \tl_case:cnn \tl_case:cnF
```

(*End definition for* \tl_case:Nnn *and* \tl_case:cnn*. These functions are documented on page* **??**.)

```
⟨/initex | package⟩
```

11 l3seq implementation

The following test files are used for this code: m3seq002,m3seq003.

```
⟨*initex | package⟩
⟨@@=seq⟩
```

A sequence is a control sequence whose top-level expansion is of the form "\s__seq __seq_item:n {⟨*item*$_1$⟩} ... __seq_item:n {⟨*item*$_n$⟩}", with a leading scan mark followed by n items of the same form. An earlier implementation used the structure "\seq_elt:w ⟨*item*$_1$⟩ \seq_elt_end: ... \seq_elt:w ⟨*item*$_n$⟩ \seq_elt_end:". This allowed rapid searching using a delimited function, but was not suitable for items containing {, } and # tokens, and also lead to the loss of surrounding braces around items.

\s__seq

The variable is defined in the l3quark module, loaded later.

(*End definition for* \s__seq. *This variable is documented on page 112.*)

__seq_item:n

The delimiter is always defined, but when used incorrectly simply removes its argument and hits an undefined control sequence to raise an error.

```
\cs_new:Npn \__seq_item:n
  {
    \__msg_kernel_expandable_error:nn { kernel } { misused-sequence }
    \use_none:n
  }
```

(*End definition for* __seq_item:n.)

\l__seq_internal_a_tl
\l__seq_internal_b_tl

Scratch space for various internal uses.

```
\tl_new:N \l__seq_internal_a_tl
\tl_new:N \l__seq_internal_b_tl
```

(*End definition for* \l__seq_internal_a_tl *and* \l__seq_internal_b_tl. *These variables are documented on page* ??.)

__seq_tmp:w

Scratch function for internal use.

```
\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.

```
\tl_const:Nn \c_empty_seq { \s__seq }
```

(*End definition for* \c_empty_seq. *This variable is documented on page 112.*)

11.1 Allocation and initialisation

\seq_new:N
\seq_new:c

Sequences are initialized to \c_empty_seq.

```
\cs_new_protected:Npn \seq_new:N #1
  {
    \__chk_if_free_cs:N #1
    \cs_gset_eq:NN #1 \c_empty_seq
  }
\cs_generate_variant:Nn \seq_new:N { c }
```

(*End definition for* \seq_new:N *and* \seq_new:c*. These functions are documented on page* **??**.)

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

Clearing a sequence is similar to setting it equal to the empty one.

```
\cs_new_protected:Npn \seq_clear:N  #1
  { \seq_set_eq:NN #1 \c_empty_seq }
\cs_generate_variant:Nn \seq_clear:N  { c }
\cs_new_protected:Npn \seq_gclear:N #1
  { \seq_gset_eq:NN #1 \c_empty_seq }
\cs_generate_variant:Nn \seq_gclear:N { c }
```

(*End definition for* \seq_clear:N *and* \seq_clear:c*. These functions are documented on page* **??**.)

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

Once again we copy code from the token list functions.

```
\cs_new_protected:Npn \seq_clear_new:N  #1
  { \seq_if_exist:NTF #1 { \seq_clear:N #1 } { \seq_new:N #1 } }
\cs_generate_variant:Nn \seq_clear_new:N  { c }
\cs_new_protected:Npn \seq_gclear_new:N #1
  { \seq_if_exist:NTF #1 { \seq_gclear:N #1 } { \seq_new:N #1 } }
\cs_generate_variant:Nn \seq_gclear_new:N { c }
```

(*End definition for* \seq_clear_new:N *and* \seq_clear_new:c*. These functions are documented on page* **??**.)

\seq_set_eq:NN
\seq_set_eq:cN
\seq_set_eq:Nc
\seq_set_eq:cc
\seq_gset_eq:NN
\seq_gset_eq:cN
\seq_gset_eq:Nc
\seq_gset_eq:cc

Copying a sequence is the same as copying the underlying token list.

```
\cs_new_eq:NN \seq_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \seq_set_eq:Nc  \tl_set_eq:Nc
\cs_new_eq:NN \seq_set_eq:cN  \tl_set_eq:cN
\cs_new_eq:NN \seq_set_eq:cc  \tl_set_eq:cc
\cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
\cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
\cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN
\cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc
```

(*End definition for* \seq_set_eq:NN *and others. These functions are documented on page* **??**.)

\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 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.

```
\cs_new_protected_nopar:Npn \seq_set_split:Nnn
  { \__seq_set_split:NNnn \tl_set:Nx }
\cs_new_protected_nopar:Npn \seq_gset_split:Nnn
  { \__seq_set_split:NNnn \tl_gset:Nx }
\cs_new_protected:Npn \__seq_set_split:NNnn #1#2#3#4
  {
    \tl_if_empty:nTF {#3}
      {
        \tl_set:Nn \l__seq_internal_a_tl
          { \tl_map_function:nN {#4} \__seq_wrap_item:n }
      }
      {
        \tl_set:Nn \l__seq_internal_a_tl
          {
            \__seq_set_split_auxi:w \prg_do_nothing:
            #4
            \__seq_set_split_end:
          }
        \tl_replace_all:Nnn \l__seq_internal_a_tl { #3 }
          {
            \__seq_set_split_end:
            \__seq_set_split_auxi:w \prg_do_nothing:
          }
        \tl_set:Nx \l__seq_internal_a_tl { \l__seq_internal_a_tl }
      }
    #1 #2 { \s__seq \l__seq_internal_a_tl }
  }
\cs_new:Npn \__seq_set_split_auxi:w #1 \__seq_set_split_end:
  {
    \exp_not:N \__seq_set_split_auxii:w
    \exp_args:No \tl_trim_spaces:n {#1}
    \exp_not:N \__seq_set_split_end:
  }
\cs_new:Npn \__seq_set_split_auxii:w #1 \__seq_set_split_end:
  { \__seq_wrap_item:n {#1} }
\cs_generate_variant:Nn \seq_set_split:Nnn  { NnV }
\cs_generate_variant:Nn \seq_gset_split:Nnn { NnV }
```

(*End definition for* `\seq_set_split:Nnn` *and others. These functions are documented on page* **??**.)

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

When concatenating sequences, one must remove the leading `\s__seq` of the second sequence. The result starts with `\s__seq` (of the first sequence), which stops f-expansion.

```
\cs_new_protected:Npn \seq_concat:NNN #1#2#3
  { \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
\cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
```

```
  { \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
\cs_generate_variant:Nn \seq_concat:NNN  { ccc }
\cs_generate_variant:Nn \seq_gconcat:NNN { ccc }
```

(*End definition for* `\seq_concat:NNN` *and* `\seq_concat:ccc`. *These functions are documented on page* ??.)

\seq_if_exist_p:N
\seq_if_exist_p:c
\seq_if_exist:NTF
\seq_if_exist:cTF

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \seq_if_exist:N \cs_if_exist:N { TF , T , F , p }
\prg_new_eq_conditional:NNn \seq_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(*End definition for* `\seq_if_exist:N` *and* `\seq_if_exist:c`. *These functions are documented on page* ??.)

11.2 Appending data to either end

\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

When adding to the left of a sequence, remove `\s__seq`. This is done by `\__seq_put_left_aux:w`, which also stops `f`-expansion.

```
\cs_new_protected:Npn \seq_put_left:Nn #1#2
  {
    \tl_set:Nx #1
      {
        \exp_not:n { \s__seq \__seq_item:n {#2} }
        \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
      }
  }
\cs_new_protected:Npn \seq_gput_left:Nn #1#2
  {
    \tl_gset:Nx #1
      {
        \exp_not:n { \s__seq \__seq_item:n {#2} }
        \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
      }
  }
\cs_new:Npn \__seq_put_left_aux:w \s__seq { \exp_stop_f: }
\cs_generate_variant:Nn \seq_put_left:Nn  {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_put_left:Nn  { c , cV , cv , co , cx }
\cs_generate_variant:Nn \seq_gput_left:Nn  {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_gput_left:Nn  { c , cV , cv , co , cx }
```

(*End definition for* `\seq_put_left:Nn` *and others. These functions are documented on page* ??.)

\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

Since there is no trailing marker, adding an item to the right of a sequence simply means wrapping it in `\__seq_item:n`.

```
\cs_new_protected:Npn \seq_put_right:Nn #1#2
  { \tl_put_right:Nn #1 { \__seq_item:n {#2} } }
\cs_new_protected:Npn \seq_gput_right:Nn #1#2
  { \tl_gput_right:Nn #1 { \__seq_item:n {#2} } }
\cs_generate_variant:Nn \seq_gput_right:Nn {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , co , cx }
\cs_generate_variant:Nn \seq_put_right:Nn {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_put_right:Nn { c , cV , cv , co , cx }
```

(*End definition for* `\seq_put_right:Nn` *and others. These functions are documented on page* ??.)

11.3 Modifying sequences

__seq_wrap_item:n

This function converts its argument to a proper sequence item in an x-expansion context.

```
\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.

```
\seq_new:N \l__seq_remove_seq
```

(*End definition for* `\l__seq_remove_seq`. *This variable is documented on page* ??.)

\seq_remove_duplicates:N
\seq_remove_duplicates:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c
__seq_remove_duplicates:NN

Removing duplicates means making a new list then copying it.

```
\cs_new_protected:Npn \seq_remove_duplicates:N
  { \__seq_remove_duplicates:NN \seq_set_eq:NN }
\cs_new_protected:Npn \seq_gremove_duplicates:N
  { \__seq_remove_duplicates:NN \seq_gset_eq:NN }
\cs_new_protected:Npn \__seq_remove_duplicates:NN #1#2
  {
    \seq_clear:N \l__seq_remove_seq
    \seq_map_inline:Nn #2
      {
        \seq_if_in:NnF \l__seq_remove_seq {##1}
          { \seq_put_right:Nn \l__seq_remove_seq {##1} }
      }
    #1 #2 \l__seq_remove_seq
  }
\cs_generate_variant:Nn \seq_remove_duplicates:N  { c }
\cs_generate_variant:Nn \seq_gremove_duplicates:N { c }
```

(*End definition for* `\seq_remove_duplicates:N` *and* `\seq_remove_duplicates:c`. *These functions are documented on page* ??.)

\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.

```
\cs_new_protected:Npn \seq_remove_all:Nn
  { \__seq_remove_all_aux:NNn \tl_set:Nx }
\cs_new_protected:Npn \seq_gremove_all:Nn
  { \__seq_remove_all_aux:NNn \tl_gset:Nx }
\cs_new_protected:Npn \__seq_remove_all_aux:NNn #1#2#3
  {
    \__seq_push_item_def:n
      {
```

```
        \str_if_eq:nnT {##1} {#3}
          {
            \if_false: { \fi: }
            \tl_set:Nn \l__seq_internal_b_tl {##1}
            #1 #2
               { \if_false: } \fi:
                 \exp_not:o {#2}
                 \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
                   { \use_none:nn }
          }
        \__seq_wrap_item:n {##1}
      }
    \tl_set:Nn \l__seq_internal_a_tl {#3}
    #1 #2 {#2}
    \__seq_pop_item_def:
  }
\cs_generate_variant:Nn \seq_remove_all:Nn  { c }
\cs_generate_variant:Nn \seq_gremove_all:Nn { c }
```

(*End definition for* `\seq_remove_all:Nn` *and* `\seq_remove_all:cn`. *These functions are documented on page* **??**.)

11.4 Sequence conditionals

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

Similar to token lists, we compare with the empty sequence.

```
\prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }
  {
    \if_meaning:w #1 \c_empty_seq
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_generate_variant:Nn \seq_if_empty_p:N { c }
\cs_generate_variant:Nn \seq_if_empty:NT { c }
\cs_generate_variant:Nn \seq_if_empty:NF { c }
\cs_generate_variant:Nn \seq_if_empty:NTF { c }
```

(*End definition for* `\seq_if_empty:N` *and* `\seq_if_empty:c`. *These functions are documented on page* **??**.)

`\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.

```
\prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
  { T , F , TF }
  {
    \group_begin:
      \tl_set:Nn \l__seq_internal_a_tl {#2}
```

```
      \cs_set_protected:Npn \__seq_item:n ##1
        {
          \tl_set:Nn \l__seq_internal_b_tl {##1}
          \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
            \exp_after:wN \__seq_if_in:
          \fi:
        }
      #1
    \group_end:
    \prg_return_false:
    \__prg_break_point:
  }
\cs_new_nopar:Npn \__seq_if_in:
  { \__prg_break:n { \group_end: \prg_return_true: } }
\cs_generate_variant:Nn \seq_if_in:NnT  {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_if_in:NnT  { c , cV , cv , co , cx }
\cs_generate_variant:Nn \seq_if_in:NnF  {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_if_in:NnF  { c , cV , cv , co , cx }
\cs_generate_variant:Nn \seq_if_in:NnTF {     NV , Nv , No , Nx }
\cs_generate_variant:Nn \seq_if_in:NnTF { c , cV , cv , co , cx }
```

(*End definition for* `\seq_if_in:NnTF` *and others. These functions are documented on page* ??.)

11.5 Recovering data from sequences

`\__seq_pop:NNNN`
`\__seq_pop_TF:NNNN`

The two `pop` functions share their emptiness tests. We also use a common emptiness test for all branching `get` and `pop` functions.

```
\cs_new_protected:Npn \__seq_pop:NNNN #1#2#3#4
  {
    \if_meaning:w #3 \c_empty_seq
      \tl_set:Nn #4 { \q_no_value }
    \else:
      #1#2#3#4
    \fi:
  }
\cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
  {
    \if_meaning:w #3 \c_empty_seq
      % \tl_set:Nn #4 { \q_no_value }
      \prg_return_false:
    \else:
      #1#2#3#4
      \prg_return_true:
    \fi:
  }
```

(*End definition for* `\__seq_pop:NNNN` *and* `\__seq_pop_TF:NNNN`*.*)

`\seq_get_left:NN`
`\seq_get_left:cN`
`\__seq_get_left:wnw`

Getting an item from the left of a sequence is pretty easy: just trim off the first item after `\__seq_item:n` at the start. We append a `\q_no_value` item to cover the case of an empty sequence

```
\cs_new_protected:Npn \seq_get_left:NN #1#2
  {
    \tl_set:Nx #2
      {
        \exp_after:wN \__seq_get_left:wnw
        #1 \__seq_item:n { \q_no_value } \q_stop
      }
  }
\cs_new:Npn \__seq_get_left:wnw #1 \__seq_item:n #2#3 \q_stop
  { \exp_not:n {#2} }
\cs_generate_variant:Nn \seq_get_left:NN { c }
```

(*End definition for* `\seq_get_left:NN` *and* `\seq_get_left:cN`*. These functions are documented on page* ??.)

`\seq_pop_left:NN`
`\seq_pop_left:cN`
`\seq_gpop_left:NN`
`\seq_gpop_left:cN`
`\__seq_pop_left:NNN`
`\__seq_pop_left:wnwNNN`

The approach to popping an item is pretty similar to that to get an item, with the only difference being that the sequence itself has to be redefined. This makes it more sensible to use an auxiliary function for the local and global cases.

```
\cs_new_protected_nopar:Npn \seq_pop_left:NN
  { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_set:Nn }
\cs_new_protected_nopar:Npn \seq_gpop_left:NN
  { \__seq_pop:NNNN \__seq_pop_left:NNN \tl_gset:Nn }
\cs_new_protected:Npn \__seq_pop_left:NNN #1#2#3
  { \exp_after:wN \__seq_pop_left:wnwNNN #2 \q_stop #1#2#3 }
\cs_new_protected:Npn \__seq_pop_left:wnwNNN
    #1 \__seq_item:n #2#3 \q_stop #4#5#6
  {
    #4 #5 { #1 #3 }
    \tl_set:Nn #6 {#2}
  }
\cs_generate_variant:Nn \seq_pop_left:NN  { c }
\cs_generate_variant:Nn \seq_gpop_left:NN { c }
```

(*End definition for* `\seq_pop_left:NN` *and* `\seq_pop_left:cN`*. These functions are documented on page* ??.)

`\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.

```
\cs_new_protected:Npn \seq_get_right:NN #1#2
  {
    \exp_after:wN \use_i_ii:nnn
    \exp_after:wN \__seq_get_right_loop:nn
    \exp_after:wN \q_no_value
    #1
    { ?? \tl_set:Nn #2 }
    { } { }
  }
```

```
\cs_new_protected:Npn \__seq_get_right_loop:nn #1#2
  {
    \use_none:nn #2 {#1}
    \__seq_get_right_loop:nn
  }
\cs_generate_variant:Nn \seq_get_right:NN { c }
```

(*End definition for* `\seq_get_right:NN` *and* `\seq_get_right:cN`*. These functions are documented on page* **??**.)

`\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 in front of the final entry. This therefore does the pop assignment. One more iteration is performed, with an empty argument and `\use_none:nn`, which finally stops the loop.

```
\cs_new_protected_nopar:Npn \seq_pop_right:NN
  { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_set:Nx }
\cs_new_protected_nopar:Npn \seq_gpop_right:NN
  { \__seq_pop:NNNN \__seq_pop_right:NNN \tl_gset:Nx }
\cs_new_protected:Npn \__seq_pop_right:NNN #1#2#3
  {
    \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
    \cs_set_eq:NN \__seq_item:n \scan_stop:
    #1 #2
      { \if_false: } \fi: \s__seq
        \exp_after:wN \use_i:nnn
        \exp_after:wN \__seq_pop_right_loop:nn
        #2
        {
          \if_false: { \fi: }
          \tl_set:Nx #3
        }
        { } \use_none:nn
    \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
  }
\cs_new:Npn \__seq_pop_right_loop:nn #1#2
  {
    #2 { \exp_not:n {#1} }
    \__seq_pop_right_loop:nn
  }
\cs_generate_variant:Nn \seq_pop_right:NN  { c }
\cs_generate_variant:Nn \seq_gpop_right:NN { c }
```

(*End definition for* `\seq_pop_right:NN` *and* `\seq_pop_right:cN`*. These functions are documented on page* **??**.)

\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.

```
\prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
  { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
\prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
  { \__seq_pop_TF:NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
\cs_generate_variant:Nn \seq_get_left:NNT   { c }
\cs_generate_variant:Nn \seq_get_left:NNF   { c }
\cs_generate_variant:Nn \seq_get_left:NNTF  { c }
\cs_generate_variant:Nn \seq_get_right:NNT  { c }
\cs_generate_variant:Nn \seq_get_right:NNF  { c }
\cs_generate_variant:Nn \seq_get_right:NNTF { c }
```

(*End definition for* \seq_get_left:NNTF *and* \seq_get_left:cNTF*. These functions are documented on page* **??**.)

\seq_pop_left:NNTF
\seq_pop_left:cNTF
\seq_gpop_left:NNTF
\seq_gpop_left:cNTF
\seq_pop_right:NNTF
\seq_pop_right:cNTF
\seq_gpop_right:NNTF
\seq_gpop_right:cNTF

More or less the same for popping.

```
\prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2 { T , F , TF }
  { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_set:Nn #1 #2 }
\prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2 { T , F , TF }
  { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_gset:Nn #1 #2 }
\prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2 { T , F , TF }
  { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \tl_set:Nx #1 #2 }
\prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2 { T , F , TF }
  { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \tl_gset:Nx #1 #2 }
\cs_generate_variant:Nn \seq_pop_left:NNT    { c }
\cs_generate_variant:Nn \seq_pop_left:NNF    { c }
\cs_generate_variant:Nn \seq_pop_left:NNTF   { c }
\cs_generate_variant:Nn \seq_gpop_left:NNT   { c }
\cs_generate_variant:Nn \seq_gpop_left:NNF   { c }
\cs_generate_variant:Nn \seq_gpop_left:NNTF  { c }
\cs_generate_variant:Nn \seq_pop_right:NNT   { c }
\cs_generate_variant:Nn \seq_pop_right:NNF   { c }
\cs_generate_variant:Nn \seq_pop_right:NNTF  { c }
\cs_generate_variant:Nn \seq_gpop_right:NNT  { c }
\cs_generate_variant:Nn \seq_gpop_right:NNF  { c }
\cs_generate_variant:Nn \seq_gpop_right:NNTF { c }
```

(*End definition for* \seq_pop_left:NNTF *and* \seq_pop_left:cNTF*. These functions are documented on page* **??**.)

11.6 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 is inserted.

```
\cs_new_nopar:Npn \seq_map_break:
  { \__prg_map_break:Nn \seq_map_break: { } }
\cs_new_nopar:Npn \seq_map_break:n
  { \__prg_map_break:Nn \seq_map_break: }
```

(*End definition for* `\seq_map_break:`. *This function is documented on page* 109.)

`\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.

```
\cs_new:Npn \seq_map_function:NN #1#2
  {
    \exp_after:wN \use_i_ii:nnn
    \exp_after:wN \__seq_map_function:NNn
    \exp_after:wN #2
    #1
    { ? \seq_map_break: } { }
    \__prg_break_point:Nn \seq_map_break: { }
  }
\cs_new:Npn \__seq_map_function:NNn #1#2#3
  {
    \use_none:n #2
    #1 {#3}
    \__seq_map_function:NNn #1
  }
\cs_generate_variant:Nn \seq_map_function:NN { c }
```

(*End definition for* `\seq_map_function:NN` *and* `\seq_map_function:cN`. *These functions are documented on page* ??.)

`\__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.

```
\cs_new_protected:Npn \__seq_push_item_def:n
  {
    \__seq_push_item_def:
    \cs_gset:Npn \__seq_item:n ##1
  }
\cs_new_protected:Npn \__seq_push_item_def:x
  {
    \__seq_push_item_def:
    \cs_gset:Npx \__seq_item:n ##1
  }
\cs_new_protected:Npn \__seq_push_item_def:
  {
    \int_gincr:N \g__prg_map_int
    \cs_gset_eq:cN { __prg_map_ \int_use:N \g__prg_map_int :w }
      \__seq_item:n
  }
\cs_new_protected_nopar:Npn \__seq_pop_item_def:
  {
    \cs_gset_eq:Nc \__seq_item:n
      { __prg_map_ \int_use:N \g__prg_map_int :w }
```

```
    \int_gdecr:N \g__prg_map_int
  }
```

(*End definition for* `\__seq_push_item_def:n` *and* `\__seq_push_item_def:x`*.*)

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

The idea here is that `\__seq_item:n` is already "applied" to each item in a sequence, and so an in-line mapping is just a case of redefining `\__seq_item:n`.

```
\cs_new_protected:Npn \seq_map_inline:Nn #1#2
  {
    \__seq_push_item_def:n {#2}
    #1
    \__prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
  }
\cs_generate_variant:Nn \seq_map_inline:Nn { c }
```

(*End definition for* `\seq_map_inline:Nn` *and* `\seq_map_inline:cn`*. These functions are documented on page* ??*.*)

`\seq_map_variable:NNn`
`\seq_map_variable:Ncn`
`\seq_map_variable:cNn`
`\seq_map_variable:ccn`

This is just a specialised version of the in-line mapping function, using an x-type expansion for the code set up so that the number of # tokens required is as expected.

```
\cs_new_protected:Npn \seq_map_variable:NNn #1#2#3
  {
    \__seq_push_item_def:x
      {
        \tl_set:Nn \exp_not:N #2 {##1}
        \exp_not:n {#3}
      }
    #1
    \__prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
  }
\cs_generate_variant:Nn \seq_map_variable:NNn {     Nc }
\cs_generate_variant:Nn \seq_map_variable:NNn { c , cc }
```

(*End definition for* `\seq_map_variable:NNn` *and others. These functions are documented on page* ??*.*)

`\seq_count:N`
`\seq_count:c`
`\__seq_count:n`

Counting the items in a sequence is done using the same approach as for other count functions: turn each entry into a +1 then use integer evaluation to actually do the mathematics.

```
\cs_new:Npn \seq_count:N #1
  {
    \int_eval:n
      {
        0
        \seq_map_function:NN #1 \__seq_count:n
      }
  }
\cs_new:Npn \__seq_count:n #1 { + \c_one }
\cs_generate_variant:Nn \seq_count:N { c }
```

(*End definition for* `\seq_count:N` *and* `\seq_count:c`*. These functions are documented on page* ??*.*)

11.7 Using sequences

\seq_use:Nnnn
\seq_use:cnnn
__seq_use:NNnNnn
__seq_use_setup:w
__seq_use:nwwwwnwn
__seq_use:nwwn
\seq_use:Nn
\seq_use:cn

See \clist_use:Nnnn for a general explanation. The main difference is that we use __seq_item:n as a delimiter rather than commas. We also need to add __seq_item:n at various places, and \s__seq.

```
\cs_new:Npn \seq_use:Nnnn #1#2#3#4
  {
    \seq_if_exist:NTF #1
      {
        \int_case:nnF { \seq_count:N #1 }
          {
            { 0 } { }
            { 1 } { \exp_after:wN \__seq_use:NNnNnn #1 ? { } { } }
            { 2 } { \exp_after:wN \__seq_use:NNnNnn #1 {#2} }
          }
          {
            \exp_after:wN \__seq_use_setup:w #1 \__seq_item:n
            \q_mark { \__seq_use:nwwwwnwn {#3} }
            \q_mark { \__seq_use:nwwn {#4} }
            \q_stop { }
          }
      }
      { \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#1} }
  }
\cs_generate_variant:Nn \seq_use:Nnnn { c }
\cs_new:Npn \__seq_use:NNnNnn #1#2#3#4#5#6 { \exp_not:n { #3 #6 #5 } }
\cs_new:Npn \__seq_use_setup:w \s__seq { \__seq_use:nwwwwnwn { } }
\cs_new:Npn \__seq_use:nwwwwnwn
    #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4#5
    \q_mark #6#7 \q_stop #8
  {
    #6 \__seq_item:n {#3} \__seq_item:n {#4} #5
    \q_mark {#6} #7 \q_stop { #8 #1 #2 }
  }
\cs_new:Npn \__seq_use:nwwn #1 \__seq_item:n #2 #3 \q_stop #4
  { \exp_not:n { #4 #1 #2 } }
\cs_new:Npn \seq_use:Nn #1#2
  { \seq_use:Nnnn #1 {#2} {#2} {#2} }
\cs_generate_variant:Nn \seq_use:Nn { c }
```

(*End definition for* \seq_use:Nnnn *and* \seq_use:cnnn*. These functions are documented on page* **??**.)

11.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

\seq_push:Nn
\seq_push:NV
\seq_push:Nv
\seq_push:No
\seq_push:Nx
\seq_push:cn
\seq_push:cV
\seq_push:cV
\seq_push:co
\seq_push:cx
\seq_gpush:Nn
\seq_gpush:NV
\seq_gpush:Nv
\seq_gpush:No

Pushing to a sequence is the same as adding on the left.

```
\cs_new_eq:NN \seq_push:Nn  \seq_put_left:Nn
\cs_new_eq:NN \seq_push:NV  \seq_put_left:NV
\cs_new_eq:NN \seq_push:Nv  \seq_put_left:Nv
```

```
\cs_new_eq:NN \seq_push:No  \seq_put_left:No
\cs_new_eq:NN \seq_push:Nx  \seq_put_left:Nx
\cs_new_eq:NN \seq_push:cn  \seq_put_left:cn
\cs_new_eq:NN \seq_push:cV  \seq_put_left:cV
\cs_new_eq:NN \seq_push:cv  \seq_put_left:cv
\cs_new_eq:NN \seq_push:co  \seq_put_left:co
\cs_new_eq:NN \seq_push:cx  \seq_put_left:cx
\cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
\cs_new_eq:NN \seq_gpush:NV \seq_gput_left:NV
\cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
\cs_new_eq:NN \seq_gpush:No \seq_gput_left:No
\cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
\cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
\cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
\cs_new_eq:NN \seq_gpush:cv \seq_gput_left:cv
\cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
\cs_new_eq:NN \seq_gpush:cx \seq_gput_left:cx
```

(*End definition for* `\seq_push:Nn` *and others. These functions are documented on page* **??**.)

\seq_get:NN
\seq_get:cN
\seq_pop:NN
\seq_pop:cN
\seq_gpop:NN
\seq_gpop:cN

In most cases, getting items from the stack does not need to specify that this is from the left. So alias are provided.

```
\cs_new_eq:NN \seq_get:NN \seq_get_left:NN
\cs_new_eq:NN \seq_get:cN \seq_get_left:cN
\cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
\cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
\cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
\cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN
```

(*End definition for* `\seq_get:NN` *and* `\seq_get:cN`*. These functions are documented on page* **??**.)

\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.

```
\prg_new_eq_conditional:NNn \seq_get:NN  \seq_get_left:NN  { T , F , TF }
\prg_new_eq_conditional:NNn \seq_get:cN  \seq_get_left:cN  { T , F , TF }
\prg_new_eq_conditional:NNn \seq_pop:NN  \seq_pop_left:NN  { T , F , TF }
\prg_new_eq_conditional:NNn \seq_pop:cN  \seq_pop_left:cN  { T , F , TF }
\prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
\prg_new_eq_conditional:NNn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }
```

(*End definition for* `\seq_get:NNTF` *and* `\seq_get:cNTF`*. These functions are documented on page* **??**.)

11.9 Viewing sequences

\seq_show:N
\seq_show:c

Apply the general `\__msg_show_variable:Nnn`.

```
\cs_new_protected:Npn \seq_show:N #1
  {
    \__msg_show_variable:Nnn #1 { seq }
      { \seq_map_function:NN #1 \__msg_show_item:n }
  }
\cs_generate_variant:Nn \seq_show:N { c }
```

(*End definition for* `\seq_show:N` *and* `\seq_show:c`*. These functions are documented on page* **??**.)

11.10 Scratch sequences

\l_tmpa_seq
\l_tmpb_seq
\g_tmpa_seq
\g_tmpb_seq

Temporary comma list variables.

```
\seq_new:N \l_tmpa_seq
\seq_new:N \l_tmpb_seq
\seq_new:N \g_tmpa_seq
\seq_new:N \g_tmpb_seq
```

(*End definition for* `\l_tmpa_seq` *and others. These variables are documented on page 112.*)

```
⟨/initex | package⟩
```

12 l3clist implementation

The following test files are used for this code: m3clist002.

```
⟨*initex | package⟩
⟨@@=clist⟩
```

\c_empty_clist

An empty comma list is simply an empty token list.

```
\cs_new_eq:NN \c_empty_clist \c_empty_tl
```

(*End definition for* `\c_empty_clist`*. This variable is documented on page 121.*)

\l__clist_internal_clist

Scratch space for various internal uses. This comma list variable cannot be declared as such because it comes before `\clist_new:N`

```
\tl_new:N \l__clist_internal_clist
```

(*End definition for* `\l__clist_internal_clist`*. This variable is documented on page* **??**.)

__clist_tmp:w

A temporary function for various purposes.

```
\cs_new_protected:Npn \__clist_tmp:w { }
```

(*End definition for* `\__clist_tmp:w`.)

12.1 Allocation and initialisation

\clist_new:N
\clist_new:c

Internally, comma lists are just token lists.

```
\cs_new_eq:NN \clist_new:N \tl_new:N
\cs_new_eq:NN \clist_new:c \tl_new:c
```

(*End definition for* `\clist_new:N` *and* `\clist_new:c`*. These functions are documented on page* **??**.)

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

Clearing comma lists is just the same as clearing token lists.

```
\cs_new_eq:NN \clist_clear:N  \tl_clear:N
\cs_new_eq:NN \clist_clear:c  \tl_clear:c
\cs_new_eq:NN \clist_gclear:N \tl_gclear:N
\cs_new_eq:NN \clist_gclear:c \tl_gclear:c
```

(*End definition for* `\clist_clear:N` *and* `\clist_clear:c`*. These functions are documented on page* **??**.)

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

Once again a copy from the token list functions.

```
\cs_new_eq:NN \clist_clear_new:N  \tl_clear_new:N
\cs_new_eq:NN \clist_clear_new:c  \tl_clear_new:c
\cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N
\cs_new_eq:NN \clist_gclear_new:c \tl_gclear_new:c
```

(*End definition for* `\clist_clear_new:N` *and* `\clist_clear_new:c`. *These functions are documented on page* **??**.)

\clist_set_eq:NN
\clist_set_eq:cN
\clist_set_eq:Nc
\clist_set_eq:cc
\clist_gset_eq:NN
\clist_gset_eq:cN
\clist_gset_eq:Nc
\clist_gset_eq:cc

Once again, these are simple copies from the token list functions.

```
\cs_new_eq:NN \clist_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \clist_set_eq:Nc  \tl_set_eq:Nc
\cs_new_eq:NN \clist_set_eq:cN  \tl_set_eq:cN
\cs_new_eq:NN \clist_set_eq:cc  \tl_set_eq:cc
\cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN
\cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc
\cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN
\cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc
```

(*End definition for* `\clist_set_eq:NN` *and others. These functions are documented on page* **??**.)

\clist_concat:NNN
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc
__clist_concat:NNNN

Concatenating comma lists is not quite as easy as it seems, as there needs to be the correct addition of a comma to the output. So a little work to do.

```
\cs_new_protected_nopar:Npn \clist_concat:NNN
  { \__clist_concat:NNNN \tl_set:Nx }
\cs_new_protected_nopar:Npn \clist_gconcat:NNN
  { \__clist_concat:NNNN \tl_gset:Nx }
\cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4
  {
    #1 #2
      {
        \exp_not:o #3
        \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
        \exp_not:o #4
      }
  }
\cs_generate_variant:Nn \clist_concat:NNN  { ccc }
\cs_generate_variant:Nn \clist_gconcat:NNN { ccc }
```

(*End definition for* `\clist_concat:NNN` *and* `\clist_concat:ccc`. *These functions are documented on page* **??**.)

\clist_if_exist_p:N
\clist_if_exist_p:c
\clist_if_exist:NTF
\clist_if_exist:cTF

Copies of the `cs` functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \clist_if_exist:N \cs_if_exist:N { TF , T , F , p }
\prg_new_eq_conditional:NNn \clist_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(*End definition for* `\clist_if_exist:N` *and* `\clist_if_exist:c`. *These functions are documented on page* **??**.)

12.2 Removing spaces around items

__clist_trim_spaces_generic:nw
__clist_trim_spaces_generic:nn

This expands to the ⟨*code*⟩, followed by a brace group containing the ⟨*item*⟩, with leading 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*⟩.

```
\cs_new:Npn \__clist_trim_spaces_generic:nw #1#2 ,
  {
    \__tl_trim_spaces:nn {#2}
      { \exp_args:No \__clist_trim_spaces_generic:nn } {#1}
  }
\cs_new:Npn \__clist_trim_spaces_generic:nn #1#2 { #2 {#1} }
```

(*End definition for* `\__clist_trim_spaces_generic:nw`.)

__clist_trim_spaces:n
__clist_trim_spaces:nn

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.

```
\cs_new:Npn \__clist_trim_spaces:n #1
  {
    \__clist_trim_spaces_generic:nw
      { \__clist_trim_spaces:nn { } }
      \q_mark #1 ,
    \q_recursion_tail, \q_recursion_stop
  }
\cs_new:Npn \__clist_trim_spaces:nn #1 #2
  {
    \quark_if_recursion_tail_stop:n {#2}
    \tl_if_empty:nTF {#2}
      {
        \__clist_trim_spaces_generic:nw
          { \__clist_trim_spaces:nn {#1} } \q_mark
      }
      {
        #1 \exp_not:n {#2}
        \__clist_trim_spaces_generic:nw
          { \__clist_trim_spaces:nn { , } } \q_mark
      }
  }
```

(*End definition for* `\__clist_trim_spaces:n`.)

12.3 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

```
\cs_new_protected:Npn \clist_set:Nn #1#2
  { \tl_set:Nx #1 { \__clist_trim_spaces:n {#2} } }
```

```
\cs_new_protected:Npn \clist_gset:Nn #1#2
  { \tl_gset:Nx #1 { \__clist_trim_spaces:n {#2} } }
\cs_generate_variant:Nn \clist_set:Nn  { NV , No , Nx , c , cV , co , cx }
\cs_generate_variant:Nn \clist_gset:Nn { NV , No , Nx , c , cV , co , cx }
```

(*End definition for* `\clist_set:Nn` *and others. These functions are documented on page* **??**.)

\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.

```
\cs_new_protected_nopar:Npn \clist_put_left:Nn
  { \__clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }
\cs_new_protected_nopar:Npn \clist_gput_left:Nn
  { \__clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
\cs_new_protected:Npn \__clist_put_left:NNNn #1#2#3#4
  {
    #2 \l__clist_internal_clist {#4}
    #1 #3 \l__clist_internal_clist #3
  }
\cs_generate_variant:Nn \clist_put_left:Nn  {     NV , No , Nx }
\cs_generate_variant:Nn \clist_put_left:Nn  { c , cV , co , cx }
\cs_generate_variant:Nn \clist_gput_left:Nn {     NV , No , Nx }
\cs_generate_variant:Nn \clist_gput_left:Nn { c , cV , co , cx }
```

(*End definition for* `\clist_put_left:Nn` *and others. These functions are documented on page* **??**.)

\clist_put_right:Nn
\clist_put_right:NV
\clist_put_right:No
\clist_put_right:Nx
\clist_put_right:cn
\clist_put_right:cV
\clist_put_right:co
\clist_put_right:cx
\clist_gput_right:Nn
\clist_gput_right:NV
\clist_gput_right:No
\clist_gput_right:Nx
\clist_gput_right:cn
\clist_gput_right:cV
\clist_gput_right:co
\clist_gput_right:cx
__clist_put_right:NNNn

```
\cs_new_protected_nopar:Npn \clist_put_right:Nn
  { \__clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
\cs_new_protected_nopar:Npn \clist_gput_right:Nn
  { \__clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
\cs_new_protected:Npn \__clist_put_right:NNNn #1#2#3#4
  {
    #2 \l__clist_internal_clist {#4}
    #1 #3 #3 \l__clist_internal_clist
  }
\cs_generate_variant:Nn \clist_put_right:Nn  {     NV , No , Nx }
\cs_generate_variant:Nn \clist_put_right:Nn  { c , cV , co , cx }
\cs_generate_variant:Nn \clist_gput_right:Nn {     NV , No , Nx }
\cs_generate_variant:Nn \clist_gput_right:Nn { c , cV , co , cx }
```

(*End definition for* `\clist_put_right:Nn` *and others. These functions are documented on page* **??**.)

12.4 Comma lists as stacks

\clist_get:NN
\clist_get:cN
__clist_get:wN

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma.

```
\cs_new_protected:Npn \clist_get:NN #1#2
  {
    \if_meaning:w #1 \c_empty_clist
      \tl_set:Nn #2 { \q_no_value }
    \else:
```

```
      \exp_after:wN \__clist_get:wN #1 , \q_stop #2
    \fi:
  }
\cs_new_protected:Npn \__clist_get:wN #1 , #2 \q_stop #3
  { \tl_set:Nn #3 {#1} }
\cs_generate_variant:Nn \clist_get:NN { c }
```

(*End definition for* `\clist_get:NN` *and* `\clist_get:cN`*. These functions are documented on page* ??.)

`\clist_pop:NN`
`\clist_pop:cN`
`\clist_gpop:NN`
`\clist_gpop:cN`
`\__clist_pop:NNN`
`\__clist_pop:wwNNN`
`\__clist_pop:wN`

An empty clist leads to `\q_no_value`, otherwise grab until the first comma and assign to the variable. The second argument of `\__clist_pop:wwNNN` is a comma list ending in a comma and `\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.

```
\cs_new_protected_nopar:Npn \clist_pop:NN
  { \__clist_pop:NNN \tl_set:Nx }
\cs_new_protected_nopar:Npn \clist_gpop:NN
  { \__clist_pop:NNN \tl_gset:Nx }
\cs_new_protected:Npn \__clist_pop:NNN #1#2#3
  {
    \if_meaning:w #2 \c_empty_clist
      \tl_set:Nn #3 { \q_no_value }
    \else:
      \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
    \fi:
  }
\cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \q_stop #3#4#5
  {
    \tl_set:Nn #5 {#1}
    #3 #4
      {
        \__clist_pop:wN \prg_do_nothing:
          #2 \exp_not:o
          , \q_mark \use_none:n
        \q_stop
      }
  }
\cs_new:Npn \__clist_pop:wN #1 , \q_mark #2 #3 \q_stop { #2 {#1} }
\cs_generate_variant:Nn \clist_pop:NN  { c }
\cs_generate_variant:Nn \clist_gpop:NN { c }
```

(*End definition for* `\clist_pop:NN` *and* `\clist_pop:cN`*. These functions are documented on page* ??.)

`\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.

```
\prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
  {
    \if_meaning:w #1 \c_empty_clist
      \prg_return_false:
    \else:
      \exp_after:wN \__clist_get:wN #1 , \q_stop #2
      \prg_return_true:
```

```
    \fi:
  }
\cs_generate_variant:Nn \clist_get:NNT  { c }
\cs_generate_variant:Nn \clist_get:NNF  { c }
\cs_generate_variant:Nn \clist_get:NNTF { c }
\prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
  { \__clist_pop_TF:NNN \tl_set:Nx #1 #2 }
\prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
  { \__clist_pop_TF:NNN \tl_gset:Nx #1 #2 }
\cs_new_protected:Npn \__clist_pop_TF:NNN #1#2#3
  {
    \if_meaning:w #2 \c_empty_clist
      \prg_return_false:
    \else:
      \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
      \prg_return_true:
    \fi:
  }
\cs_generate_variant:Nn \clist_pop:NNT   { c }
\cs_generate_variant:Nn \clist_pop:NNF   { c }
\cs_generate_variant:Nn \clist_pop:NNTF  { c }
\cs_generate_variant:Nn \clist_gpop:NNT  { c }
\cs_generate_variant:Nn \clist_gpop:NNF  { c }
\cs_generate_variant:Nn \clist_gpop:NNTF { c }
```

(*End definition for* `\clist_get:NNTF` *and* `\clist_get:cNTF`. *These functions are documented on page* ??.)

\clist_push:Nn
\clist_push:NV
\clist_push:No
\clist_push:Nx
\clist_push:cn
\clist_push:cV
\clist_push:co
\clist_push:cx
\clist_gpush:Nn
\clist_gpush:NV
\clist_gpush:No
\clist_gpush:Nx
\clist_gpush:cn
\clist_gpush:cV
\clist_gpush:co
\clist_gpush:cx

Pushing to a comma list is the same as adding on the left.

```
\cs_new_eq:NN \clist_push:Nn  \clist_put_left:Nn
\cs_new_eq:NN \clist_push:NV  \clist_put_left:NV
\cs_new_eq:NN \clist_push:No  \clist_put_left:No
\cs_new_eq:NN \clist_push:Nx  \clist_put_left:Nx
\cs_new_eq:NN \clist_push:cn  \clist_put_left:cn
\cs_new_eq:NN \clist_push:cV  \clist_put_left:cV
\cs_new_eq:NN \clist_push:co  \clist_put_left:co
\cs_new_eq:NN \clist_push:cx  \clist_put_left:cx
\cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn
\cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV
\cs_new_eq:NN \clist_gpush:No \clist_gput_left:No
\cs_new_eq:NN \clist_gpush:Nx \clist_gput_left:Nx
\cs_new_eq:NN \clist_gpush:cn \clist_gput_left:cn
\cs_new_eq:NN \clist_gpush:cV \clist_gput_left:cV
\cs_new_eq:NN \clist_gpush:co \clist_gput_left:co
\cs_new_eq:NN \clist_gpush:cx \clist_gput_left:cx
```

(*End definition for* `\clist_push:Nn` *and others. These functions are documented on page* ??.)

12.5 Modifying comma lists

\l__clist_internal_remove_clist

An internal comma list for the removal routines.

```
\clist_new:N \l__clist_internal_remove_clist
```

(*End definition for* `\l__clist_internal_remove_clist`. *This variable is documented on page* ??.)

`\clist_remove_duplicates:N`
`\clist_remove_duplicates:c`
`\clist_gremove_duplicates:N`
`\clist_gremove_duplicates:c`
`\__clist_remove_duplicates:NN`

Removing duplicates means making a new list then copying it.

```
\cs_new_protected:Npn \clist_remove_duplicates:N
  { \__clist_remove_duplicates:NN \clist_set_eq:NN }
\cs_new_protected:Npn \clist_gremove_duplicates:N
  { \__clist_remove_duplicates:NN \clist_gset_eq:NN }
\cs_new_protected:Npn \__clist_remove_duplicates:NN #1#2
  {
    \clist_clear:N \l__clist_internal_remove_clist
    \clist_map_inline:Nn #2
      {
        \clist_if_in:NnF \l__clist_internal_remove_clist {##1}
          { \clist_put_right:Nn \l__clist_internal_remove_clist {##1} }
      }
    #1 #2 \l__clist_internal_remove_clist
  }
\cs_generate_variant:Nn \clist_remove_duplicates:N  { c }
\cs_generate_variant:Nn \clist_gremove_duplicates:N { c }
```

(*End definition for* `\clist_remove_duplicates:N` *and* `\clist_remove_duplicates:c`. *These functions are documented on page* ??.)

`\clist_remove_all:Nn`
`\clist_remove_all:cn`
`\clist_gremove_all:Nn`
`\clist_gremove_all:cn`
`\__clist_remove_all:NNn`
`\__clist_remove_all:w`
`\__clist_remove_all:`

The method used here is very similar to `\tl_replace_all:Nnn`. Build a function delimited by the ⟨*item*⟩ that should be removed, surrounded with commas, and call that function followed by the expanded comma list, and another copy of the ⟨*item*⟩. The loop is controlled by the argument grabbed by `\__clist_remove_all:w`: when the item was found, the `\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.

```
\cs_new_protected:Npn \clist_remove_all:Nn
  { \__clist_remove_all:NNn \tl_set:Nx }
\cs_new_protected:Npn \clist_gremove_all:Nn
  { \__clist_remove_all:NNn \tl_gset:Nx }
\cs_new_protected:Npn \__clist_remove_all:NNn #1#2#3
  {
    \cs_set:Npn \__clist_tmp:w ##1 , #3 ,
      {
        ##1
        , \q_mark , \use_none_delimit_by_q_stop:w ,
        \__clist_remove_all:
```

```
      }
    #1 #2
      {
        \exp_after:wN \__clist_remove_all:
        #2 , \q_mark , #3 , \q_stop
      }
    \clist_if_empty:NF #2
      {
        #1 #2
          {
            \exp_args:No \exp_not:o
              { \exp_after:wN \use_none:n #2 }
          }
      }
  }
\cs_new:Npn \__clist_remove_all:
  { \exp_after:wN \__clist_remove_all:w \__clist_tmp:w , }
\cs_new:Npn \__clist_remove_all:w #1 , \q_mark , #2 , { \exp_not:n {#1} }
\cs_generate_variant:Nn \clist_remove_all:Nn  { c }
\cs_generate_variant:Nn \clist_gremove_all:Nn { c }
```

(*End definition for* `\clist_remove_all:Nn` *and* `\clist_remove_all:cn`*. These functions are documented on page* ??.)

12.6 Comma list conditionals

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

Simple copies from the token list variable material.

```
\prg_new_eq_conditional:NNn \clist_if_empty:N \tl_if_empty:N { p , T , F , TF }
\prg_new_eq_conditional:NNn \clist_if_empty:c \tl_if_empty:c { p , T , F , TF }
```

(*End definition for* `\clist_if_empty:N` *and* `\clist_if_empty:c`*. These functions are documented on page* ??.)

`\clist_if_in:NnTF`
`\clist_if_in:NVTF`
`\clist_if_in:NoTF`
`\clist_if_in:cnTF`
`\clist_if_in:cVTF`
`\clist_if_in:coTF`
`\clist_if_in:nnTF`
`\clist_if_in:nVTF`
`\clist_if_in:noTF`
`\__clist_if_in_return:nn`

See description of the `\tl_if_in:Nn` function for details. We simply surround the comma list, and the item, with commas.

```
\prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T  , F , TF }
  {
    \exp_args:No \__clist_if_in_return:nn #1 {#2}
  }
\prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T  , F , TF }
  {
    \clist_set:Nn \l__clist_internal_clist {#1}
    \exp_args:No \__clist_if_in_return:nn \l__clist_internal_clist {#2}
  }
\cs_new_protected:Npn \__clist_if_in_return:nn #1#2
  {
    \cs_set:Npn \__clist_tmp:w ##1 ,#2, { }
    \tl_if_empty:oTF
      { \__clist_tmp:w ,#1, {} {} ,#2, }
      { \prg_return_false: } { \prg_return_true: }
  }
```

```
\cs_generate_variant:Nn \clist_if_in:NnT  {     NV , No }
\cs_generate_variant:Nn \clist_if_in:NnT  { c , cV , co }
\cs_generate_variant:Nn \clist_if_in:NnF  {     NV , No }
\cs_generate_variant:Nn \clist_if_in:NnF  { c , cV , co }
\cs_generate_variant:Nn \clist_if_in:NnTF {     NV , No }
\cs_generate_variant:Nn \clist_if_in:NnTF { c , cV , co }
\cs_generate_variant:Nn \clist_if_in:nnT  {     nV , no }
\cs_generate_variant:Nn \clist_if_in:nnF  {     nV , no }
\cs_generate_variant:Nn \clist_if_in:nnTF {     nV , no }
```

(*End definition for* `\clist_if_in:NnTF` *and others. These functions are documented on page* **??**.)

12.7 Mapping to comma lists

`\clist_map_function:NN`
`\clist_map_function:cN`
`\__clist_map_function:Nw`

If the variable is empty, the mapping is skipped (otherwise, that comma-list would be seen as consisting of one empty item). Then loop over the comma-list, grabbing one comma-delimited item at a time. The end is marked by `\q_recursion_tail`. The auxiliary function `\__clist_map_function:Nw` is used directly in `\clist_map_inline:Nn`. Change with care.

```
\cs_new:Npn \clist_map_function:NN #1#2
  {
    \clist_if_empty:NF #1
      {
        \exp_last_unbraced:NNo \__clist_map_function:Nw #2 #1
          , \q_recursion_tail ,
        \__prg_break_point:Nn \clist_map_break: { }
      }
  }
\cs_new:Npn \__clist_map_function:Nw #1#2 ,
  {
    \__quark_if_recursion_tail_break:nN {#2} \clist_map_break:
    #1 {#2}
    \__clist_map_function:Nw #1
  }
\cs_generate_variant:Nn \clist_map_function:NN { c }
```

(*End definition for* `\clist_map_function:NN` *and* `\clist_map_function:cN`*. These functions are documented on page* **??**.)

`\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`.

```
\cs_new:Npn \clist_map_function:nN #1#2
  {
    \__clist_trim_spaces_generic:nw { \__clist_map_function_n:Nn #2 }
    \q_mark #1, \q_recursion_tail,
    \__prg_break_point:Nn \clist_map_break: { }
```

```
  }
\cs_new:Npn \__clist_map_function_n:Nn #1 #2
  {
    \__quark_if_recursion_tail_break:nN {#2} \clist_map_break:
    \tl_if_empty:nF {#2} { \__clist_map_unbrace:Nw #1 #2, }
    \__clist_trim_spaces_generic:nw { \__clist_map_function_n:Nn #1 }
    \q_mark
  }
\cs_new:Npn \__clist_map_unbrace:Nw #1 #2, { #1 {#2} }
```

(*End definition for* `\clist_map_function:nN`*. This function is documented on page* **??**.)

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

Inline mapping is done by creating a suitable function "on the fly": this is done globally to avoid any issues with TEX's groups. We use a different function for each level of nesting.

Since the mapping is non-expandable, we can perform the space-trimming needed by the `n` version simply by storing the comma-list in a variable. We don't need a different comma-list for each nesting level: the comma-list is expanded before the mapping starts.

```
\cs_new_protected:Npn \clist_map_inline:Nn #1#2
  {
    \clist_if_empty:NF #1
      {
        \int_gincr:N \g__prg_map_int
        \cs_gset:cpn { __prg_map_ \int_use:N \g__prg_map_int :w } ##1 {#2}
        \exp_last_unbraced:Nco \__clist_map_function:Nw
          { __prg_map_ \int_use:N \g__prg_map_int :w }
          #1 , \q_recursion_tail ,
        \__prg_break_point:Nn \clist_map_break:
          { \int_gdecr:N \g__prg_map_int }
      }
  }
\cs_new_protected:Npn \clist_map_inline:nn #1
  {
    \clist_set:Nn \l__clist_internal_clist {#1}
    \clist_map_inline:Nn \l__clist_internal_clist
  }
\cs_generate_variant:Nn \clist_map_inline:Nn { c }
```

(*End definition for* `\clist_map_inline:Nn` *and* `\clist_map_inline:cn`*. These functions are documented on page* **??**.)

`\clist_map_variable:NNn`
`\clist_map_variable:cNn`
`\clist_map_variable:nNn`
`\__clist_map_variable:Nnw`

As for other comma-list mappings, filter out the case of an empty list. Same approach as `\clist_map_function:Nn`, additionally we store each item in the given variable. As for inline mappings, space trimming for the `n` variant is done by storing the comma list in a variable.

```
\cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
  {
    \clist_if_empty:NF #1
      {
        \exp_args:Nno \use:nn
          { \__clist_map_variable:Nnw #2 {#3} }
```

```
          #1
          , \q_recursion_tail , \q_recursion_stop
        \__prg_break_point:Nn \clist_map_break: { }
      }
  }
\cs_new_protected:Npn \clist_map_variable:nNn #1
  {
    \clist_set:Nn \l__clist_internal_clist {#1}
    \clist_map_variable:NNn \l__clist_internal_clist
  }
\cs_new_protected:Npn \__clist_map_variable:Nnw #1#2#3,
  {
    \tl_set:Nn #1 {#3}
    \quark_if_recursion_tail_stop:N #1
    \use:n {#2}
    \__clist_map_variable:Nnw #1 {#2}
  }
\cs_generate_variant:Nn \clist_map_variable:NNn { c }
```

(*End definition for* `\clist_map_variable:NNn` *and* `\clist_map_variable:cNn`*. These functions are documented on page* **??**.)

`\clist_map_break:`
`\clist_map_break:n`

The break statements use the general `\__prg_map_break:Nn` mechanism.

```
\cs_new_nopar:Npn \clist_map_break:
  { \__prg_map_break:Nn \clist_map_break: { } }
\cs_new_nopar:Npn \clist_map_break:n
  { \__prg_map_break:Nn \clist_map_break: }
```

(*End definition for* `\clist_map_break:` *and* `\clist_map_break:n`*. These functions are documented on page 118.*)

`\clist_count:N`
`\clist_count:c`
`\clist_count:n`
`\__clist_count:n`
`\__clist_count:w`

Counting the items in a comma list is done using the same approach as for other token count functions: turn each entry into a `+1` then use integer evaluation to actually do the mathematics. In the case of an `n`-type comma-list, we could of course use `\clist_map_function:nN`, but that is very slow, because it carefully removes spaces. Instead, we loop manually, and skip blank items (but not `{}`, hence the extra spaces).

```
\cs_new:Npn \clist_count:N #1
  {
    \int_eval:n
      {
        0
        \clist_map_function:NN #1 \__clist_count:n
      }
  }
\cs_generate_variant:Nn \clist_count:N { c }
\cs_new:Npx \clist_count:n #1
  {
    \exp_not:N \int_eval:n
      {
        0
        \exp_not:N \__clist_count:w \c_space_tl
```

```
      #1 \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
    }
  }
\cs_new:Npn \__clist_count:n #1 { + \c_one }
\cs_new:Npx \__clist_count:w #1 ,
  {
    \exp_not:n { \exp_args:Nf \quark_if_recursion_tail_stop:n } {#1}
    \exp_not:N \tl_if_blank:nF {#1} { + \c_one }
    \exp_not:N \__clist_count:w \c_space_tl
  }
```

(*End definition for* `\clist_count:N`*,* `\clist_count:c`*, and* `\clist_count:n`*. These functions are documented on page* **??***.*)

12.8 Using comma lists

`\clist_use:Nnnn`
`\clist_use:cnnn`
`\__clist_use:wwn`
`\__clist_use:nwwwwnwn`
`\__clist_use:nwwn`
`\clist_use:Nn`
`\clist_use:cn`

First check that the variable exists. Then count the items in the comma list. If it has none, output nothing. If it has one item, output that item, brace stripped (note that space-trimming has already been done when the comma list was assigned). If it has two, place the ⟨*separator between two*⟩ in the middle.

Otherwise, `\__clist_use:nwwwwnwn` takes the following arguments; 1: a ⟨*separator*⟩, 2, 3, 4: three items from the comma list (or quarks), 5: the rest of the comma list, 6: a ⟨*continuation*⟩ function (`use_ii` or `use_iii` with its ⟨*separator*⟩ argument), 7: junk, and 8: the temporary result, which is built in a brace group following `\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 `\q_mark` serves as a delimiter to `use_ii`, switching to the other ⟨*continuation*⟩, `use_iii`, which uses the ⟨*separator between final two*⟩.

```
\cs_new:Npn \clist_use:Nnnn #1#2#3#4
  {
    \clist_if_exist:NTF #1
      {
        \int_case:nnF { \clist_count:N #1 }
          {
            { 0 } { }
            { 1 } { \exp_after:wN \__clist_use:wwn #1 , , { } }
            { 2 } { \exp_after:wN \__clist_use:wwn #1 , {#2} }
          }
          {
            \exp_after:wN \__clist_use:nwwwwnwn
            \exp_after:wN { \exp_after:wN } #1 ,
            \q_mark , { \__clist_use:nwwwwnwn {#3} }
            \q_mark , { \__clist_use:nwwn {#4} }
            \q_stop { }
          }
      }
```

```
      { \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#1} }
  }
\cs_generate_variant:Nn \clist_use:Nnnn { c }
\cs_new:Npn \__clist_use:wwn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
\cs_new:Npn \__clist_use:nwwwwnwn
    #1#2 , #3 , #4 , #5 \q_mark , #6#7 \q_stop #8
  { #6 {#3} , {#4} , #5 \q_mark , {#6} #7 \q_stop { #8 #1 #2 } }
\cs_new:Npn \__clist_use:nwwn #1#2 , #3 \q_stop #4
  { \exp_not:n { #4 #1 #2 } }
\cs_new:Npn \clist_use:Nn #1#2
  { \clist_use:Nnnn #1 {#2} {#2} {#2} }
\cs_generate_variant:Nn \clist_use:Nn { c }
```

(*End definition for* `\clist_use:Nnnn` *and* `\clist_use:cnnn`*. These functions are documented on page* ??.)

12.9 Viewing comma lists

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

Apply the general `\__msg_show_variable:Nnn`. In the case of an `n`-type comma-list, first store it in a scratch variable, then show that variable: The message takes care of omitting its name.

```
\cs_new_protected:Npn \clist_show:N #1
  {
    \__msg_show_variable:Nnn #1 { clist }
      { \clist_map_function:NN #1 \__msg_show_item:n }
  }
\cs_new_protected:Npn \clist_show:n #1
  {
    \clist_set:Nn \l__clist_internal_clist {#1}
    \clist_show:N \l__clist_internal_clist
  }
\cs_generate_variant:Nn \clist_show:N { c }
```

(*End definition for* `\clist_show:N` *and* `\clist_show:c`*. These functions are documented on page 121.*)

12.10 Scratch comma lists

`\l_tmpa_clist`
`\l_tmpb_clist`
`\g_tmpa_clist`
`\g_tmpb_clist`

Temporary comma list variables.

```
\clist_new:N \l_tmpa_clist
\clist_new:N \l_tmpb_clist
\clist_new:N \g_tmpa_clist
\clist_new:N \g_tmpb_clist
```

(*End definition for* `\l_tmpa_clist` *and* `\l_tmpb_clist`*. These variables are documented on page 121.*)

```
⟨/initex | package⟩
```

13 l3prop implementation

The following test files are used for this code: m3prop001, m3prop002, m3prop003, m3prop004, m3show001.

```
⟨*initex | package⟩
⟨@@=prop⟩
```

A property list is a macro whose top-level expansion is for the form

\s__prop __prop_pair:wn ⟨key_1⟩ \s__prop {⟨$value_1$⟩}
...
__prop_pair:wn ⟨key_n⟩ \s__prop {⟨$value_n$⟩}

where \s__prop is a scan mark (equal to \scan_stop:), 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.

```
\__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.

```
\cs_new:Npn \__prop_pair:wn #1 \s__prop #2
  { \__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.

```
\tl_new:N \l__prop_internal_tl
```

(*End definition for* \l__prop_internal_tl*. This variable is documented on page 127.*)

\c_empty_prop

An empty prop.

```
\tl_const:Nn \c_empty_prop { \s__prop }
```

(*End definition for* \c_empty_prop*. This variable is documented on page 127.*)

13.1 Allocation and initialisation

\prop_new:N
\prop_new:c

Property lists are initialized with the value \c_empty_prop.

```
\cs_new_protected:Npn \prop_new:N #1
  {
    \__chk_if_free_cs:N #1
    \cs_gset_eq:NN #1 \c_empty_prop
  }
\cs_generate_variant:Nn \prop_new:N { c }
```

(*End definition for* \prop_new:N *and* \prop_new:c*. These functions are documented on page* **??***.*)

\prop_clear:N
\prop_clear:c
\prop_gclear:N
\prop_gclear:c

The same idea for clearing.

```
\cs_new_protected:Npn \prop_clear:N  #1
  { \prop_set_eq:NN #1 \c_empty_prop }
\cs_generate_variant:Nn \prop_clear:N  { c }
\cs_new_protected:Npn \prop_gclear:N #1
  { \prop_gset_eq:NN #1 \c_empty_prop }
\cs_generate_variant:Nn \prop_gclear:N { c }
```

(*End definition for* `\prop_clear:N` *and* `\prop_clear:c`*. These functions are documented on page* **??**.)

\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.

```
\cs_new_protected:Npn \prop_clear_new:N  #1
  { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
\cs_generate_variant:Nn \prop_clear_new:N  { c }
\cs_new_protected:Npn \prop_gclear_new:N #1
  { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
\cs_generate_variant:Nn \prop_gclear_new:N { c }
```

(*End definition for* `\prop_clear_new:N` *and* `\prop_clear_new:c`*. These functions are documented on page* **??**.)

\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.

```
\cs_new_eq:NN \prop_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \prop_set_eq:Nc  \tl_set_eq:Nc
\cs_new_eq:NN \prop_set_eq:cN  \tl_set_eq:cN
\cs_new_eq:NN \prop_set_eq:cc  \tl_set_eq:cc
\cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
\cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
\cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
\cs_new_eq:NN \prop_gset_eq:cc \tl_gset_eq:cc
```

(*End definition for* `\prop_set_eq:NN` *and others. These functions are documented on page* **??**.)

\l_tmpa_prop
\l_tmpb_prop
\g_tmpa_prop
\g_tmpb_prop

We can now initialize the scratch variables.

```
\prop_new:N \l_tmpa_prop
\prop_new:N \l_tmpb_prop
\prop_new:N \g_tmpa_prop
\prop_new:N \g_tmpb_prop
```

(*End definition for* `\l_tmpa_prop` *and* `\l_tmpb_prop`*. These variables are documented on page* *127*.)

13.2 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 ⟨$extract_1$⟩ before the key–value pair, the ⟨*value*⟩ associated with the ⟨*key*⟩ and the ⟨$extract_2$⟩ 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*⟩.

```
\cs_new_protected:Npn \__prop_split:NnTF #1#2
  { \exp_args:NNo \__prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }
\cs_new_protected:Npn \__prop_split_aux:NnTF #1#2#3#4
  {
    \cs_set:Npn \__prop_split_aux:w ##1
      \__prop_pair:wn #2 \s__prop ##2 ##3 \q_mark ##4 ##5 \q_stop
      { ##4 {#3} {#4} }
    \exp_after:wN \__prop_split_aux:w #1 \q_mark \use_i:nn
      \__prop_pair:wn #2 \s__prop { } \q_mark \use_ii:nn \q_stop
  }
\cs_new:Npn \__prop_split_aux:w { }
```

(*End definition for* `\__prop_split:NnTF`.)

`\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.

```
\cs_new_protected:Npn \prop_remove:Nn #1#2
  {
    \__prop_split:NnTF #1 {#2}
      { \tl_set:Nn #1 { ##1 ##3 } }
      { }
  }
\cs_new_protected:Npn \prop_gremove:Nn #1#2
  {
    \__prop_split:NnTF #1 {#2}
      { \tl_gset:Nn #1 { ##1 ##3 } }
      { }
  }
\cs_generate_variant:Nn \prop_remove:Nn  {     NV }
\cs_generate_variant:Nn \prop_remove:Nn  { c , cV }
\cs_generate_variant:Nn \prop_gremove:Nn {     NV }
\cs_generate_variant:Nn \prop_gremove:Nn { c , cV }
```

(*End definition for* `\prop_remove:Nn` *and others. These functions are documented on page* **??**.)

`\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`.

```
\cs_new_protected:Npn \prop_get:NnN #1#2#3
  {
    \__prop_split:NnTF #1 {#2}
      { \tl_set:Nn #3 {##2} }
      { \tl_set:Nn #3 { \q_no_value } }
```

```
  }
\cs_generate_variant:Nn \prop_get:NnN {     NV , No }
\cs_generate_variant:Nn \prop_get:NnN { c , cV , co }
```

(*End definition for* `\prop_get:NnN` *and others. These functions are documented on page* **??**.)

`\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.

```
\cs_new_protected:Npn \prop_pop:NnN #1#2#3
  {
    \__prop_split:NnTF #1 {#2}
      {
        \tl_set:Nn #3 {##2}
        \tl_set:Nn #1 { ##1 ##3 }
      }
      { \tl_set:Nn #3 { \q_no_value } }
  }
\cs_new_protected:Npn \prop_gpop:NnN #1#2#3
  {
    \__prop_split:NnTF #1 {#2}
      {
        \tl_set:Nn #3 {##2}
        \tl_gset:Nn #1 { ##1 ##3 }
      }
      { \tl_set:Nn #3 { \q_no_value } }
  }
\cs_generate_variant:Nn \prop_pop:NnN  {     No }
\cs_generate_variant:Nn \prop_pop:NnN  { c , co }
\cs_generate_variant:Nn \prop_gpop:NnN {     No }
\cs_generate_variant:Nn \prop_gpop:NnN { c , co }
```

(*End definition for* `\prop_pop:NnN` *and others. These functions are documented on page* **??**.)

`\prop_pop:NnNTF`
`\prop_pop:cnNTF`
`\prop_gpop:NnNTF`
`\prop_gpop:cnNTF`

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.

```
\prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
  {
    \__prop_split:NnTF #1 {#2}
      {
        \tl_set:Nn #3 {##2}
        \tl_set:Nn #1 { ##1 ##3 }
        \prg_return_true:
      }
      { \prg_return_false: }
  }
\prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
  {
    \__prop_split:NnTF #1 {#2}
      {
```

```
          \tl_set:Nn #3 {##2}
          \tl_gset:Nn #1 { ##1 ##3 }
          \prg_return_true:
        }
        { \prg_return_false: }
    }
\cs_generate_variant:Nn \prop_pop:NnNT   { c }
\cs_generate_variant:Nn \prop_pop:NnNF   { c }
\cs_generate_variant:Nn \prop_pop:NnNTF  { c }
\cs_generate_variant:Nn \prop_gpop:NnNT  { c }
\cs_generate_variant:Nn \prop_gpop:NnNF  { c }
\cs_generate_variant:Nn \prop_gpop:NnNTF { c }
```

(*End definition for* `\prop_pop:NnNTF` *and others. These functions are documented on page* **??**.)

\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.

```
\cs_new_protected_nopar:Npn \prop_put:Nnn  { \__prop_put:NNnn \tl_set:Nx }
\cs_new_protected_nopar:Npn \prop_gput:Nnn { \__prop_put:NNnn \tl_gset:Nx }
\cs_new_protected:Npn \__prop_put:NNnn #1#2#3#4
  {
    \tl_set:Nn \l__prop_internal_tl
      {
        \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
        \s__prop { \exp_not:n {#4} }
      }
    \__prop_split:NnTF #2 {#3}
      { #1 #2 { \exp_not:n {##1} \l__prop_internal_tl \exp_not:n {##3} } }
      { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
  }
\cs_generate_variant:Nn \prop_put:Nnn
  {     NnV , Nno , Nnx , NV , NVV , No , Noo }
\cs_generate_variant:Nn \prop_put:Nnn
  { c , cnV , cno , cnx , cV , cVV , co , coo }
\cs_generate_variant:Nn \prop_gput:Nnn
  {     NnV , Nno , Nnx , NV , NVV , No , Noo }
\cs_generate_variant:Nn \prop_gput:Nnn
  { c , cnV , cno , cnx , cV , cVV , co , coo }
```

(*End definition for* `\prop_put:Nnn` *and others. These functions are documented on page* **??**.)

\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`.

```
\cs_new_protected_nopar:Npn \prop_put_if_new:Nnn
  { \__prop_put_if_new:NNnn \tl_set:Nx }
\cs_new_protected_nopar:Npn \prop_gput_if_new:Nnn
  { \__prop_put_if_new:NNnn \tl_gset:Nx }
\cs_new_protected:Npn \__prop_put_if_new:NNnn #1#2#3#4
  {
    \tl_set:Nn \l__prop_internal_tl
      {
        \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
        \s__prop \exp_not:n { {#4} }
      }
    \__prop_split:NnTF #2 {#3}
      { }
      { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
  }
\cs_generate_variant:Nn \prop_put_if_new:Nnn  { c }
\cs_generate_variant:Nn \prop_gput_if_new:Nnn { c }
```

(*End definition for `\prop_put_if_new:Nnn` and `\prop_put_if_new:cnn`. These functions are documented on page* **??**.)

13.3 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.

```
\prg_new_eq_conditional:NNn \prop_if_exist:N \cs_if_exist:N { TF , T , F , p }
\prg_new_eq_conditional:NNn \prop_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(*End definition for `\prop_if_exist:N` and `\prop_if_exist:c`. These functions are documented on page* **??**.)

\prop_if_empty_p:N
\prop_if_empty_p:c
\prop_if_empty:NTF
\prop_if_empty:cTF

Same test as for token lists.

```
\prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
  {
    \tl_if_eq:NNTF #1 \c_empty_prop
      \prg_return_true: \prg_return_false:
  }
\cs_generate_variant:Nn \prop_if_empty_p:N { c }
\cs_generate_variant:Nn \prop_if_empty:NT { c }
\cs_generate_variant:Nn \prop_if_empty:NF { c }
\cs_generate_variant:Nn \prop_if_empty:NTF { c }
```

(*End definition for `\prop_if_empty:N` and `\prop_if_empty:c`. These functions are documented on page* **??**.)

\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.

```
\prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
  {
    \exp_last_unbraced:Noo \__prop_if_in:nwwn { \tl_to_str:n {#2} } #1
      \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
      \q_recursion_tail
    \__prg_break_point:
  }
\cs_new:Npn \__prop_if_in:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
  {
    \str_if_eq_x:nnTF {#1} {#3}
      { \__prop_if_in:N }
      { \__prop_if_in:nwwn {#1} }
  }
\cs_new:Npn \__prop_if_in:N #1
  {
    \if_meaning:w \q_recursion_tail #1
      \prg_return_false:
    \else:
      \prg_return_true:
    \fi:
    \__prg_break:
  }
\cs_generate_variant:Nn \prop_if_in_p:Nn {     NV , No }
\cs_generate_variant:Nn \prop_if_in_p:Nn { c , cV , co }
\cs_generate_variant:Nn \prop_if_in:NnT  {     NV , No }
\cs_generate_variant:Nn \prop_if_in:NnT  { c , cV , co }
\cs_generate_variant:Nn \prop_if_in:NnF  {     NV , No }
\cs_generate_variant:Nn \prop_if_in:NnF  { c , cV , co }
\cs_generate_variant:Nn \prop_if_in:NnTF {     NV , No }
\cs_generate_variant:Nn \prop_if_in:NnTF { c , cV , co }
```

(*End definition for* \prop_if_in:Nn *and others. These functions are documented on page* ??.)

13.4 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.

```
\prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
  {
    \__prop_split:NnTF #1 {#2}
      {
        \tl_set:Nn #3 {##2}
        \prg_return_true:
      }
      { \prg_return_false: }
  }
\cs_generate_variant:Nn \prop_get:NnNT  {     NV , No }
\cs_generate_variant:Nn \prop_get:NnNF  {     NV , No }
\cs_generate_variant:Nn \prop_get:NnNTF {     NV , No }
\cs_generate_variant:Nn \prop_get:NnNT  { c , cV , co }
\cs_generate_variant:Nn \prop_get:NnNF  { c , cV , co }
\cs_generate_variant:Nn \prop_get:NnNTF { c , cV , co }
```

(*End definition for* `\prop_get:NnNTF` *and others. These functions are documented on page* **??**.)

13.5 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`.

```
\cs_new:Npn \prop_map_function:NN #1#2
  {
    \exp_last_unbraced:NNo \__prop_map_function:Nwwn #2 #1
      \__prop_pair:wn \q_recursion_tail \s__prop { }
    \__prg_break_point:Nn \prop_map_break: { }
  }
\cs_new:Npn \__prop_map_function:Nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
  {
    \if_meaning:w \q_recursion_tail #3
      \exp_after:wN \prop_map_break:
    \fi:
    #1 {#3} {#4}
    \__prop_map_function:Nwwn #1
  }
\cs_generate_variant:Nn \prop_map_function:NN {     Nc }
\cs_generate_variant:Nn \prop_map_function:NN { c , cc }
```

(*End definition for* `\prop_map_function:NN` *and others. These functions are documented on page* **??**.)

`\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.

```
\cs_new_protected:Npn \prop_map_inline:Nn #1#2
  {
```

```
    \cs_gset_eq:cN
      { __prg_map_ \int_use:N \g__prg_map_int :wn } \__prop_pair:wn
    \int_gincr:N \g__prg_map_int
    \cs_gset:Npn \__prop_pair:wn ##1 \s__prop ##2 {#2}
    #1
    \__prg_break_point:Nn \prop_map_break:
      {
        \int_gdecr:N \g__prg_map_int
        \cs_gset_eq:Nc \__prop_pair:wn
          { __prg_map_ \int_use:N \g__prg_map_int :wn }
      }
  }
\cs_generate_variant:Nn \prop_map_inline:Nn { c }
```

(*End definition for* `\prop_map_inline:Nn` *and* `\prop_map_inline:cn`*. These functions are documented on page* ??.)

`\prop_map_break:`
`\prop_map_break:n`

The break statements are based on the general `\__prg_map_break:Nn`.

```
\cs_new_nopar:Npn \prop_map_break:
  { \__prg_map_break:Nn \prop_map_break: { } }
\cs_new_nopar:Npn \prop_map_break:n
  { \__prg_map_break:Nn \prop_map_break: }
```

(*End definition for* `\prop_map_break:`*. This function is documented on page 126.*)

13.6 Viewing property lists

`\prop_show:N`
`\prop_show:c`

Apply the general `\__msg_show_variable:Nnn`. Contrarily to sequences and comma lists, we use `\__msg_show_item:nn` to format both the key and the value for each pair.

```
\cs_new_protected:Npn \prop_show:N #1
  {
    \__msg_show_variable:Nnn #1 { prop }
      { \prop_map_function:NN #1 \__msg_show_item:nn }
  }
\cs_generate_variant:Nn \prop_show:N { c }
```

(*End definition for* `\prop_show:N` *and* `\prop_show:c`*. These functions are documented on page* ??.)

```
⟨/initex | package⟩
```

14 l3box implementation

```
⟨*initex | package⟩
```

```
⟨@@=box⟩
```

The code in this module is very straight forward so I'm not going to comment it very extensively.

14.1 Creating and initialising boxes

The following test files are used for this code: m3box001.lvt.

\box_new:N
\box_new:c

Defining a new ⟨*box*⟩ register: remember that box 255 is not generally available.

```
⟨*package⟩
\cs_new_protected:Npn \box_new:N #1
  {
    \__chk_if_free_cs:N #1
    \cs:w newbox \cs_end: #1
  }
⟨/package⟩
\cs_generate_variant:Nn \box_new:N { c }
```

\box_clear:N
\box_clear:c
\box_gclear:N
\box_gclear:c

Clear a ⟨*box*⟩ register.

```
\cs_new_protected:Npn \box_clear:N #1
  { \box_set_eq:NN  #1 \c_empty_box }
\cs_new_protected:Npn \box_gclear:N #1
  { \box_gset_eq:NN #1 \c_empty_box }
\cs_generate_variant:Nn \box_clear:N  { c }
\cs_generate_variant:Nn \box_gclear:N { c }
```

\box_clear_new:N
\box_clear_new:c
\box_gclear_new:N
\box_gclear_new:c

Clear or new.

```
\cs_new_protected:Npn \box_clear_new:N #1
  { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
\cs_new_protected:Npn \box_gclear_new:N #1
  { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
\cs_generate_variant:Nn \box_clear_new:N  { c }
\cs_generate_variant:Nn \box_gclear_new:N { c }
```

\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

Assigning the contents of a box to be another box.

```
\cs_new_protected:Npn \box_set_eq:NN #1#2
  { \tex_setbox:D #1 \tex_copy:D #2 }
\cs_new_protected:Npn \box_gset_eq:NN
  { \tex_global:D \box_set_eq:NN }
\cs_generate_variant:Nn \box_set_eq:NN  { c , Nc , cc }
\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).

```
\cs_new_protected:Npn \box_set_eq_clear:NN #1#2
  { \tex_setbox:D #1 \tex_box:D #2 }
\cs_new_protected:Npn \box_gset_eq_clear:NN
  { \tex_global:D  \box_set_eq_clear:NN }
\cs_generate_variant:Nn \box_set_eq_clear:NN  { c , Nc , cc }
\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:N*TF*
\box_if_exist:c*TF*

Copies of the cs functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N { TF , T , F , p }
\prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

14.2 Measuring and setting box dimensions

\box_ht:N
\box_ht:c
\box_dp:N
\box_dp:c
\box_wd:N
\box_wd:c

Accessing the height, depth, and width of a ⟨*box*⟩ register.

```
\cs_new_eq:NN \box_ht:N \tex_ht:D
\cs_new_eq:NN \box_dp:N \tex_dp:D
\cs_new_eq:NN \box_wd:N \tex_wd:D
\cs_generate_variant:Nn \box_ht:N { c }
\cs_generate_variant:Nn \box_dp:N { c }
\cs_generate_variant:Nn \box_wd:N { c }
```

\box_set_ht:Nn
\box_set_ht:cn
\box_set_dp:Nn
\box_set_dp:cn
\box_set_wd:Nn
\box_set_wd:cn

Measuring is easy: all primitive work. These primitives are not expandable, so the derived functions are not either.

```
\cs_new_protected:Npn \box_set_dp:Nn #1#2
  { \box_dp:N #1 \__dim_eval:w #2 \__dim_eval_end: }
\cs_new_protected:Npn \box_set_ht:Nn #1#2
  { \box_ht:N #1 \__dim_eval:w #2 \__dim_eval_end: }
\cs_new_protected:Npn \box_set_wd:Nn #1#2
  { \box_wd:N #1 \__dim_eval:w #2 \__dim_eval_end: }
\cs_generate_variant:Nn \box_set_ht:Nn { c }
\cs_generate_variant:Nn \box_set_dp:Nn { c }
\cs_generate_variant:Nn \box_set_wd:Nn { c }
```

14.3 Using boxes

\box_use_clear:N
\box_use_clear:c
\box_use:N
\box_use:c

Using a ⟨*box*⟩. These are just TeX primitives with meaningful names.

```
\cs_new_eq:NN \box_use_clear:N \tex_box:D
\cs_new_eq:NN \box_use:N \tex_copy:D
\cs_generate_variant:Nn \box_use_clear:N { c }
\cs_generate_variant:Nn \box_use:N { c }
```

\box_move_left:nn
\box_move_right:nn
\box_move_up:nn
\box_move_down:nn

Move box material in different directions.

```
\cs_new_protected:Npn \box_move_left:nn #1#2
  { \tex_moveleft:D \__dim_eval:w #1 \__dim_eval_end: #2 }
\cs_new_protected:Npn \box_move_right:nn #1#2
  { \tex_moveright:D \__dim_eval:w #1 \__dim_eval_end: #2 }
\cs_new_protected:Npn \box_move_up:nn #1#2
  { \tex_raise:D \__dim_eval:w #1 \__dim_eval_end: #2 }
\cs_new_protected:Npn \box_move_down:nn #1#2
  { \tex_lower:D \__dim_eval:w #1 \__dim_eval_end: #2 }
```

14.4 Box conditionals

\if_hbox:N
\if_vbox:N
\if_box_empty:N

The primitives for testing if a ⟨*box*⟩ is empty/void or which type of box it is.

```
\cs_new_eq:NN \if_hbox:N      \tex_ifhbox:D
\cs_new_eq:NN \if_vbox:N      \tex_ifvbox:D
\cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D
```

\box_if_horizontal_p:N
\box_if_horizontal_p:c
\box_if_horizontal:NTF
\box_if_horizontal:cTF
\box_if_vertical_p:N
\box_if_vertical_p:c
\box_if_vertical:NTF
\box_if_vertical:cTF

```
\prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
  { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
  { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\cs_generate_variant:Nn \box_if_horizontal_p:N { c }
\cs_generate_variant:Nn \box_if_horizontal:NT  { c }
\cs_generate_variant:Nn \box_if_horizontal:NF  { c }
\cs_generate_variant:Nn \box_if_horizontal:NTF { c }
\cs_generate_variant:Nn \box_if_vertical_p:N { c }
\cs_generate_variant:Nn \box_if_vertical:NT  { c }
\cs_generate_variant:Nn \box_if_vertical:NF  { c }
\cs_generate_variant:Nn \box_if_vertical:NTF { c }
```

\box_if_empty_p:N
\box_if_empty_p:c
\box_if_empty:NTF
\box_if_empty:cTF

Testing if a ⟨*box*⟩ is empty/void.

```
\prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
  { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\cs_generate_variant:Nn \box_if_empty_p:N { c }
\cs_generate_variant:Nn \box_if_empty:NT  { c }
\cs_generate_variant:Nn \box_if_empty:NF  { c }
\cs_generate_variant:Nn \box_if_empty:NTF { c }
```

(*End definition for* `\box_new:N` *and* `\box_new:c`*. These functions are documented on page* **??**.)

14.5 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.

```
\cs_new_protected:Npn \box_set_to_last:N #1
  { \tex_setbox:D #1 \tex_lastbox:D }
\cs_new_protected:Npn \box_gset_to_last:N
  { \tex_global:D \box_set_to_last:N }
\cs_generate_variant:Nn \box_set_to_last:N  { c }
\cs_generate_variant:Nn \box_gset_to_last:N { c }
```

(*End definition for* `\box_set_to_last:N` *and* `\box_set_to_last:c`*. These functions are documented on page* **??**.)

14.6 Constant boxes

\c_empty_box

A box we never use.

```
\box_new:N \c_empty_box
```

(*End definition for* `\c_empty_box`*. This variable is documented on page 131.*)

14.7 Scratch boxes

\l_tmpa_box
\l_tmpb_box
\g_tmpa_box
\g_tmpb_box

Scratch boxes.

```
\box_new:N \l_tmpa_box
\box_new:N \l_tmpb_box
\box_new:N \g_tmpa_box
\box_new:N \g_tmpb_box
```

(*End definition for* `\l_tmpa_box` *and others. These variables are documented on page 131.*)

14.8 Viewing box contents

TeX's `\tex_showbox:D` 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.

```
\cs_new_protected:Npn \box_show:N #1
  { \box_show:Nnn #1 \c_max_int \c_max_int }
\cs_generate_variant:Nn \box_show:N { c }
\cs_new_protected_nopar:Npn \box_show:Nnn
  { \__box_show:NNnn \c_one }
\cs_generate_variant:Nn \box_show:Nnn { c }
```

(*End definition for* `\box_show:N` *and* `\box_show:c`*. These functions are documented on page* ??.)

`\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.

```
\cs_new_protected:Npn \box_log:N #1
  { \box_log:Nnn #1 \c_max_int \c_max_int }
\cs_generate_variant:Nn \box_log:N { c }
\cs_new_protected:Npn \box_log:Nnn #1#2#3
  {
    \use:x
      {
        \etex_interactionmode:D \c_zero
        \__box_show:NNnn \c_zero \exp_not:N #1
          { \int_eval:n {#2} } { \int_eval:n {#3} }
        \etex_interactionmode:D
            = \tex_the:D \etex_interactionmode:D \scan_stop:
      }
  }
\cs_generate_variant:Nn \box_log:Nnn { c }
```

(*End definition for* `\box_log:N` *and* `\box_log:c`*. These functions are documented on page* ??.)

`\__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 `\tex_tracingonline:D` is used to control what appears in the terminal.

```
\cs_new_protected:Npn \__box_show:NNnn #1#2#3#4
  {
    \group_begin:
      \int_set:Nn \tex_showboxbreadth:D {#3}
      \int_set:Nn \tex_showboxdepth:D   {#4}
      \int_set_eq:NN \tex_tracingonline:D #1
      \box_if_exist:NTF #2
        { \tex_showbox:D \use:n {#2} }
        {
          \__msg_kernel_error:nnx { kernel } { variable-not-defined }
            { \token_to_str:N #2 }
        }
```

```
    \group_end:
  }
```

(*End definition for* `\__box_show:NNnn`.)

14.9 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.

```
\cs_new_protected:Npn \hbox:n { \tex_hbox:D \scan_stop: }
```

(*End definition for* `\hbox:n`. *This function is documented on page* 132.)

`\hbox_set:Nn`
`\hbox_set:cn`
`\hbox_gset:Nn`
`\hbox_gset:cn`

```
\cs_new_protected:Npn \hbox_set:Nn #1#2 { \tex_setbox:D #1 \tex_hbox:D {#2} }
\cs_new_protected:Npn \hbox_gset:Nn { \tex_global:D \hbox_set:Nn }
\cs_generate_variant:Nn \hbox_set:Nn { c }
\cs_generate_variant:Nn \hbox_gset:Nn { c }
```

(*End definition for* `\hbox_set:Nn` *and* `\hbox_set:cn`. *These functions are documented on page* ??.)

`\hbox_set_to_wd:Nnn`
`\hbox_set_to_wd:cnn`
`\hbox_gset_to_wd:Nnn`
`\hbox_gset_to_wd:cnn`

Storing material in a horizontal box with a specified width.

```
\cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
  { \tex_setbox:D #1 \tex_hbox:D to \__dim_eval:w #2 \__dim_eval_end: {#3} }
\cs_new_protected:Npn \hbox_gset_to_wd:Nnn
  { \tex_global:D \hbox_set_to_wd:Nnn }
\cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
\cs_generate_variant:Nn \hbox_gset_to_wd:Nnn { c }
```

(*End definition for* `\hbox_set_to_wd:Nnn` *and* `\hbox_set_to_wd:cnn`. *These functions are documented on page* ??.)

`\hbox_set:Nw`
`\hbox_set:cw`
`\hbox_gset:Nw`
`\hbox_gset:cw`
`\hbox_set_end:`
`\hbox_gset_end:`

Storing material in a horizontal box. This type is useful in environment definitions.

```
\cs_new_protected:Npn \hbox_set:Nw  #1
  { \tex_setbox:D #1 \tex_hbox:D \c_group_begin_token }
\cs_new_protected:Npn \hbox_gset:Nw
  { \tex_global:D \hbox_set:Nw }
\cs_generate_variant:Nn \hbox_set:Nw  { c }
\cs_generate_variant:Nn \hbox_gset:Nw { c }
\cs_new_eq:NN \hbox_set_end:  \c_group_end_token
\cs_new_eq:NN \hbox_gset_end: \c_group_end_token
```

(*End definition for* `\hbox_set:Nw` *and* `\hbox_set:cw`. *These functions are documented on page* 133.)

`\hbox_set_inline_begin:N`
`\hbox_set_inline_begin:c`
`\hbox_gset_inline_begin:N`
`\hbox_gset_inline_begin:c`
`\hbox_set_inline_end:`
`\hbox_gset_inline_end:`

Renamed September 2011.

```
\cs_new_eq:NN \hbox_set_inline_begin:N  \hbox_set:Nw
\cs_new_eq:NN \hbox_set_inline_begin:c  \hbox_set:cw
\cs_new_eq:NN \hbox_set_inline_end:     \hbox_set_end:
\cs_new_eq:NN \hbox_gset_inline_begin:N \hbox_gset:Nw
\cs_new_eq:NN \hbox_gset_inline_begin:c \hbox_gset:cw
\cs_new_eq:NN \hbox_gset_inline_end:    \hbox_gset_end:
```

(*End definition for* `\hbox_set_inline_begin:N` *and* `\hbox_set_inline_begin:c`. *These functions are documented on page* ??.)

\hbox_to_wd:nn
\hbox_to_zero:n

Put a horizontal box directly into the input stream.

```
\cs_new_protected:Npn \hbox_to_wd:nn #1#2
   { \tex_hbox:D to \__dim_eval:w #1 \__dim_eval_end: {#2} }
\cs_new_protected:Npn \hbox_to_zero:n #1 { \tex_hbox:D to \c_zero_dim {#1} }
```

(*End definition for* `\hbox_to_wd:nn`*. This function is documented on page 132.*)

\hbox_overlap_left:n
\hbox_overlap_right:n

Put a zero-sized box with the contents pushed against one side (which makes it stick out on the other) directly into the input stream.

```
\cs_new_protected:Npn \hbox_overlap_left:n  #1
  { \hbox_to_zero:n { \tex_hss:D #1 } }
\cs_new_protected:Npn \hbox_overlap_right:n #1
  { \hbox_to_zero:n { #1 \tex_hss:D } }
```

(*End definition for* `\hbox_overlap_left:n` *and* `\hbox_overlap_right:n`*. These functions are documented on page 132.*)

\hbox_unpack:N
\hbox_unpack:c
\hbox_unpack_clear:N
\hbox_unpack_clear:c

Unpacking a box and if requested also clear it.

```
\cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
\cs_new_eq:NN \hbox_unpack_clear:N \tex_unhbox:D
\cs_generate_variant:Nn \hbox_unpack:N { c }
\cs_generate_variant:Nn \hbox_unpack_clear:N { c }
```

(*End definition for* `\hbox_unpack:N` *and* `\hbox_unpack:c`*. These functions are documented on page* **??**.)

14.10 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.

\vbox_top:n

The following test files are used for this code: m3box003.lvt.

Put a vertical box directly into the input stream.

```
\cs_new_protected:Npn \vbox:n #1     { \tex_vbox:D { #1 \par } }
\cs_new_protected:Npn \vbox_top:n #1 { \tex_vtop:D { #1 \par } }
```

(*End definition for* `\vbox:n`*. This function is documented on page 133.*)

\vbox_to_ht:nn
\vbox_to_zero:n
\vbox_to_ht:nn
\vbox_to_zero:n

Put a vertical box directly into the input stream.

```
\cs_new_protected:Npn \vbox_to_ht:nn #1#2
  { \tex_vbox:D to \__dim_eval:w #1 \__dim_eval_end: { #2 \par } }
\cs_new_protected:Npn \vbox_to_zero:n #1
  { \tex_vbox:D to \c_zero_dim { #1 \par } }
```

(*End definition for* `\vbox_to_ht:nn` *and* `\vbox_to_zero:n`*. These functions are documented on page 134.*)

\vbox_set:Nn
\vbox_set:cn
\vbox_gset:Nn
\vbox_gset:cn

Storing material in a vertical box with a natural height.

```
\cs_new_protected:Npn \vbox_set:Nn #1#2
  { \tex_setbox:D #1 \tex_vbox:D { #2 \par } }
\cs_new_protected:Npn \vbox_gset:Nn  { \tex_global:D \vbox_set:Nn }
\cs_generate_variant:Nn \vbox_set:Nn  { c }
\cs_generate_variant:Nn \vbox_gset:Nn { c }
```

(*End definition for* `\vbox_set:Nn` *and* `\vbox_set:cn`*. These functions are documented on page* **??**.)

`\vbox_set_top:Nn`
`\vbox_set_top:cn`
`\vbox_gset_top:Nn`
`\vbox_gset_top:cn`

Storing material in a vertical box with a natural height and reference point at the baseline of the first object in the box.

```
\cs_new_protected:Npn \vbox_set_top:Nn #1#2
  { \tex_setbox:D #1 \tex_vtop:D { #2 \par } }
\cs_new_protected:Npn \vbox_gset_top:Nn
  { \tex_global:D \vbox_set_top:Nn }
\cs_generate_variant:Nn \vbox_set_top:Nn { c }
\cs_generate_variant:Nn \vbox_gset_top:Nn { c }
```

(*End definition for* `\vbox_set_top:Nn` *and* `\vbox_set_top:cn`*. These functions are documented on page* **??**.)

`\vbox_set_to_ht:Nnn`
`\vbox_set_to_ht:cnn`
`\vbox_gset_to_ht:Nnn`
`\vbox_gset_to_ht:cnn`

Storing material in a vertical box with a specified height.

```
\cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3
  { \tex_setbox:D #1 \tex_vbox:D to \__dim_eval:w #2 \__dim_eval_end: { #3 \par } }
\cs_new_protected:Npn \vbox_gset_to_ht:Nnn
  { \tex_global:D \vbox_set_to_ht:Nnn }
\cs_generate_variant:Nn \vbox_set_to_ht:Nnn  { c }
\cs_generate_variant:Nn \vbox_gset_to_ht:Nnn { c }
```

(*End definition for* `\vbox_set_to_ht:Nnn` *and* `\vbox_set_to_ht:cnn`*. These functions are documented on page* **??**.)

`\vbox_set:Nw`
`\vbox_set:cw`
`\vbox_gset:Nw`
`\vbox_gset:cw`
`\vbox_set_end:`
`\vbox_gset_end:`

Storing material in a vertical box. This type is useful in environment definitions.

```
\cs_new_protected:Npn \vbox_set:Nw #1
  { \tex_setbox:D #1 \tex_vbox:D \c_group_begin_token }
\cs_new_protected:Npn \vbox_gset:Nw
  { \tex_global:D \vbox_set:Nw }
\cs_generate_variant:Nn \vbox_set:Nw  { c }
\cs_generate_variant:Nn \vbox_gset:Nw { c }
\cs_new_protected:Npn \vbox_set_end:
  {
    \par
    \c_group_end_token
  }
\cs_new_eq:NN \vbox_gset_end: \vbox_set_end:
```

(*End definition for* `\vbox_set:Nw` *and* `\vbox_set:cw`*. These functions are documented on page* *134*.)

`\vbox_set_inline_begin:N`
`\vbox_set_inline_begin:c`
`\vbox_gset_inline_begin:N`
`\vbox_gset_inline_begin:c`
`\vbox_set_inline_end:`
`\vbox_gset_inline_end:`

Renamed September 2011.

```
\cs_new_eq:NN \vbox_set_inline_begin:N  \vbox_set:Nw
\cs_new_eq:NN \vbox_set_inline_begin:c  \vbox_set:cw
\cs_new_eq:NN \vbox_set_inline_end:  \vbox_set_end:
\cs_new_eq:NN \vbox_gset_inline_begin:N \vbox_gset:Nw
\cs_new_eq:NN \vbox_gset_inline_begin:c \vbox_gset:cw
\cs_new_eq:NN \vbox_gset_inline_end: \vbox_gset_end:
```

(*End definition for* `\vbox_set_inline_begin:N` *and* `\vbox_set_inline_begin:c`*. These functions are documented on page* **??**.)

\vbox_unpack:N
\vbox_unpack:c
\vbox_unpack_clear:N
\vbox_unpack_clear:c

Unpacking a box and if requested also clear it.

```
\cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
\cs_new_eq:NN \vbox_unpack_clear:N \tex_unvbox:D
\cs_generate_variant:Nn \vbox_unpack:N { c }
\cs_generate_variant:Nn \vbox_unpack_clear:N { c }
```

(*End definition for* `\vbox_unpack:N` *and* `\vbox_unpack:c`. *These functions are documented on page* ??.)

\vbox_set_split_to_ht:NNn

Splitting a vertical box in two.

```
\cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
  { \tex_setbox:D #1 \tex_vsplit:D #2 to \__dim_eval:w #3 \__dim_eval_end: }
```

(*End definition for* `\vbox_set_split_to_ht:NNn`. *This function is documented on page 134.*)

```
⟨/initex | package⟩
```

15 l3coffins Implementation

```
⟨*initex | package⟩
⟨@@=coffin⟩
```

15.1 Coffins: data structures and general variables

\l__coffin_internal_box
\l__coffin_internal_dim
\l__coffin_internal_tl

Scratch variables.

```
\box_new:N \l__coffin_internal_box
\dim_new:N \l__coffin_internal_dim
\tl_new:N  \l__coffin_internal_tl
```

(*End definition for* `\l__coffin_internal_box`. *This variable is documented on page* ??.)

\c__coffin_corners_prop

The "corners"; of a coffin define the real content, as opposed to the TeX bounding box. They all start off in the same place, of course.

```
\prop_new:N \c__coffin_corners_prop
\prop_put:Nnn \c__coffin_corners_prop { tl } { { 0 pt } { 0 pt } }
\prop_put:Nnn \c__coffin_corners_prop { tr } { { 0 pt } { 0 pt } }
\prop_put:Nnn \c__coffin_corners_prop { bl } { { 0 pt } { 0 pt } }
\prop_put:Nnn \c__coffin_corners_prop { br } { { 0 pt } { 0 pt } }
```

(*End definition for* `\c__coffin_corners_prop`. *This variable is documented on page* ??.)

\c__coffin_poles_prop

Pole positions are given for horizontal, vertical and reference-point based values.

```
\prop_new:N \c__coffin_poles_prop
\tl_set:Nn \l__coffin_internal_tl { { 0 pt } { 0 pt } { 0 pt } { 1000 pt } }
\prop_put:Nno \c__coffin_poles_prop { l }  { \l__coffin_internal_tl }
\prop_put:Nno \c__coffin_poles_prop { hc } { \l__coffin_internal_tl }
\prop_put:Nno \c__coffin_poles_prop { r }  { \l__coffin_internal_tl }
\tl_set:Nn \l__coffin_internal_tl { { 0 pt } { 0 pt } { 1000 pt } { 0 pt } }
\prop_put:Nno \c__coffin_poles_prop { b }  { \l__coffin_internal_tl }
\prop_put:Nno \c__coffin_poles_prop { vc } { \l__coffin_internal_tl }
\prop_put:Nno \c__coffin_poles_prop { t }  { \l__coffin_internal_tl }
\prop_put:Nno \c__coffin_poles_prop { B }  { \l__coffin_internal_tl }
\prop_put:Nno \c__coffin_poles_prop { H }  { \l__coffin_internal_tl }
\prop_put:Nno \c__coffin_poles_prop { T }  { \l__coffin_internal_tl }
```

(*End definition for* `\c__coffin_poles_prop`*. This variable is documented on page* **??**.)

`\l__coffin_slope_x_fp`
`\l__coffin_slope_y_fp`

Used for calculations of intersections.

```
\fp_new:N \l__coffin_slope_x_fp
\fp_new:N \l__coffin_slope_y_fp
```

(*End definition for* `\l__coffin_slope_x_fp`*. This variable is documented on page* **??**.)

`\l__coffin_error_bool`

For propagating errors so that parts of the code can work around them.

```
\bool_new:N \l__coffin_error_bool
```

(*End definition for* `\l__coffin_error_bool`*. This variable is documented on page* **??**.)

`\l__coffin_offset_x_dim`
`\l__coffin_offset_y_dim`

The offset between two sets of coffin handles when typesetting. These values are corrected from those requested in an alignment for the positions of the handles.

```
\dim_new:N \l__coffin_offset_x_dim
\dim_new:N \l__coffin_offset_y_dim
```

(*End definition for* `\l__coffin_offset_x_dim`*. This variable is documented on page* **??**.)

`\l__coffin_pole_a_tl`
`\l__coffin_pole_b_tl`

Needed for finding the intersection of two poles.

```
\tl_new:N \l__coffin_pole_a_tl
\tl_new:N \l__coffin_pole_b_tl
```

(*End definition for* `\l__coffin_pole_a_tl`*. This variable is documented on page* **??**.)

`\l__coffin_x_dim`
`\l__coffin_y_dim`
`\l__coffin_x_prime_dim`
`\l__coffin_y_prime_dim`

For calculating intersections and so forth.

```
\dim_new:N \l__coffin_x_dim
\dim_new:N \l__coffin_y_dim
\dim_new:N \l__coffin_x_prime_dim
\dim_new:N \l__coffin_y_prime_dim
```

(*End definition for* `\l__coffin_x_dim`*. This variable is documented on page* **??**.)

15.2 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:NTF`
`\coffin_if_exist:cTF`

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.

```
\prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }
  {
    \cs_if_exist:NTF #1
      {
        \cs_if_exist:cTF { l__coffin_poles_ \__int_value:w #1 _prop }
          { \prg_return_true: }
          { \prg_return_false: }
      }
      { \prg_return_false: }
  }
\cs_generate_variant:Nn \coffin_if_exist_p:N { c }
```

```
\cs_generate_variant:Nn \coffin_if_exist:NT  { c }
\cs_generate_variant:Nn \coffin_if_exist:NF  { c }
\cs_generate_variant:Nn \coffin_if_exist:NTF { c }
```

(*End definition for* `\coffin_if_exist:N` *and* `\coffin_if_exist:c`*. These functions are documented on page* **??**.)

`\__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.

```
\cs_new_protected:Npn \__coffin_if_exist:NT #1#2
  {
    \coffin_if_exist:NTF #1
      { #2 }
      {
        \__msg_kernel_error:nnx { kernel } { unknown-coffin }
          { \token_to_str:N #1 }
      }
  }
```

(*End definition for* `\__coffin_if_exist:NT`*. This function is documented on page* **??**.)

`\coffin_clear:N`
`\coffin_clear:c`

Clearing coffins means emptying the box and resetting all of the structures.

```
\cs_new_protected:Npn \coffin_clear:N #1
  {
    \__coffin_if_exist:NT #1
      {
        \box_clear:N #1
        \__coffin_reset_structure:N #1
      }
  }
\cs_generate_variant:Nn \coffin_clear:N { c }
```

(*End definition for* `\coffin_clear:N` *and* `\coffin_clear:c`*. These functions are documented on page* **??**.)

`\coffin_new:N`
`\coffin_new:c`

Creating a new coffin means making the underlying box and adding the data structures. 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.

```
\cs_new_protected:Npn \coffin_new:N #1
  {
    \box_new:N #1
    \prop_clear_new:c { l__coffin_corners_ \__int_value:w #1 _prop }
    \prop_clear_new:c { l__coffin_poles_   \__int_value:w #1 _prop }
    \prop_gset_eq:cN { l__coffin_corners_ \__int_value:w #1 _prop }
      \c__coffin_corners_prop
    \prop_gset_eq:cN { l__coffin_poles_ \__int_value:w #1 _prop }
      \c__coffin_poles_prop
  }
\cs_generate_variant:Nn \coffin_new:N { c }
```

(*End definition for* `\coffin_new:N` *and* `\coffin_new:c`*. These functions are documented on page* **??**.)

\hcoffin_set:Nn
\hcoffin_set:cn

Horizontal coffins are relatively easy: set the appropriate box, reset the structures then update the handle positions.

```
\cs_new_protected:Npn \hcoffin_set:Nn #1#2
  {
    \__coffin_if_exist:NT #1
      {
        \hbox_set:Nn #1
          {
            \color_group_begin:
              \color_ensure_current:
              #2
            \color_group_end:
          }
        \__coffin_reset_structure:N #1
        \__coffin_update_poles:N #1
        \__coffin_update_corners:N #1
      }
  }
\cs_generate_variant:Nn \hcoffin_set:Nn { c }
```

(*End definition for `\hcoffin_set:Nn` and `\hcoffin_set:cn`. These functions are documented on page* **??**.)

\vcoffin_set:Nnn
\vcoffin_set:cnn

Setting vertical coffins is more complex. First, the material is typeset with a given width. The default handles and poles are set as for a horizontal coffin, before finding the top baseline using a temporary box. No `\color_ensure_current:` here as that would add a whatsit to the start of the vertical box and mess up the location of the `T` pole (see *TeX by Topic* for discussion of the `\vtop` primitive, used to do the measuring).

```
\cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
  {
    \__coffin_if_exist:NT #1
      {
        \vbox_set:Nn #1
          {
            \dim_set:Nn \tex_hsize:D {#2}
⟨*package⟩
            \dim_set_eq:NN \linewidth   \tex_hsize:D
            \dim_set_eq:NN \columnwidth \tex_hsize:D
⟨/package⟩
            \color_group_begin:
              #3
            \color_group_end:
          }
        \__coffin_reset_structure:N #1
        \__coffin_update_poles:N #1
        \__coffin_update_corners:N #1
        \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
        \__coffin_set_pole:Nnx #1 { T }
          {
            { 0 pt }
```

```
          { \dim_eval:n { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box } }
          { 1000 pt }
          { 0 pt }
        }
      \box_clear:N \l__coffin_internal_box
    }
  }
\cs_generate_variant:Nn \vcoffin_set:Nnn { c }
```

(*End definition for* `\vcoffin_set:Nnn` *and* `\vcoffin_set:cnn`*. These functions are documented on page* ??.)

\hcoffin_set:Nw
\hcoffin_set:cw
\hcoffin_set_end:

These are the "begin"/"end" versions of the above: watch the grouping!

```
\cs_new_protected:Npn \hcoffin_set:Nw #1
  {
    \__coffin_if_exist:NT #1
      {
        \hbox_set:Nw #1 \color_group_begin: \color_ensure_current:
          \cs_set_protected_nopar:Npn \hcoffin_set_end:
            {
                \color_group_end:
              \hbox_set_end:
              \__coffin_reset_structure:N #1
              \__coffin_update_poles:N #1
              \__coffin_update_corners:N #1
            }
      }
  }
\cs_new_protected_nopar:Npn \hcoffin_set_end: { }
\cs_generate_variant:Nn \hcoffin_set:Nw { c }
```

(*End definition for* `\hcoffin_set:Nw` *and* `\hcoffin_set:cw`*. These functions are documented on page 137.*)

\vcoffin_set:Nnw
\vcoffin_set:cnw
\vcoffin_set_end:

The same for vertical coffins.

```
\cs_new_protected:Npn \vcoffin_set:Nnw #1#2
  {
    \__coffin_if_exist:NT #1
      {
        \vbox_set:Nw #1
          \dim_set:Nn \tex_hsize:D {#2}
⟨*package⟩
            \dim_set_eq:NN \linewidth   \tex_hsize:D
            \dim_set_eq:NN \columnwidth \tex_hsize:D
⟨/package⟩
          \color_group_begin: \color_ensure_current:
          \cs_set_protected:Npn \vcoffin_set_end:
            {
                \color_group_end:
              \vbox_set_end:
              \__coffin_reset_structure:N #1
```

```
                \__coffin_update_poles:N #1
                \__coffin_update_corners:N #1
                \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
                \__coffin_set_pole:Nnx #1 { T }
                  {
                    { 0 pt }
                    {
                      \dim_eval:n { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
                    }
                    { 1000 pt }
                    { 0 pt }
                  }
                \box_clear:N \l__coffin_internal_box
              }
        }
    }
\cs_new_protected_nopar:Npn \vcoffin_set_end: { }
\cs_generate_variant:Nn \vcoffin_set:Nnw { c }
```

(*End definition for* `\vcoffin_set:Nnw` *and* `\vcoffin_set:cnw`*. These functions are documented on page 137.*)

`\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.

```
\cs_new_protected:Npn \coffin_set_eq:NN #1#2
  {
    \__coffin_if_exist:NT #1
      {
        \box_set_eq:NN #1 #2
        \__coffin_set_eq_structure:NN #1 #2
      }
  }
\cs_generate_variant:Nn \coffin_set_eq:NN { c , Nc , cc }
```

(*End definition for* `\coffin_set_eq:NN` *and others. These functions are documented on page* **??**.)

`\c_empty_coffin`
`\l__coffin_aligned_coffin`
`\l__coffin_aligned_internal_coffin`

Special coffins: these cannot be set up earlier as they need `\coffin_new:N`. The empty coffin is set as a box as the full coffin-setting system needs some material which is not yet available.

```
\coffin_new:N \c_empty_coffin
\hbox_set:Nn  \c_empty_coffin { }
\coffin_new:N \l__coffin_aligned_coffin
\coffin_new:N \l__coffin_aligned_internal_coffin
```

(*End definition for* `\c_empty_coffin`*. This variable is documented on page* **??**.)

`\l_tmpa_coffin`
`\l_tmpb_coffin`

The usual scratch space.

```
\coffin_new:N \l_tmpa_coffin
\coffin_new:N \l_tmpb_coffin
```

(*End definition for* `\l_tmpa_coffin` *and* `\l_tmpb_coffin`*. These variables are documented on page 139.*)

15.3 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.

```
\cs_new_eq:NN \coffin_dp:N \box_dp:N
\cs_new_eq:NN \coffin_dp:c \box_dp:c
\cs_new_eq:NN \coffin_ht:N \box_ht:N
\cs_new_eq:NN \coffin_ht:c \box_ht:c
\cs_new_eq:NN \coffin_wd:N \box_wd:N
\cs_new_eq:NN \coffin_wd:c \box_wd:c
```

(*End definition for* `\coffin_dp:N` *and others. These functions are documented on page* **??**.)

15.4 Coffins: handle and pole management

__coffin_get_pole:NnN

A simple wrapper around the recovery of a coffin pole, with some error checking and recovery built-in.

```
\cs_new_protected:Npn \__coffin_get_pole:NnN #1#2#3
  {
    \prop_get:cnNF
      { l__coffin_poles_ \__int_value:w #1 _prop } {#2} #3
      {
        \__msg_kernel_error:nnxx { kernel } { unknown-coffin-pole }
          {#2} { \token_to_str:N #1 }
        \tl_set:Nn #3 { { 0 pt } { 0 pt } { 0 pt } { 0 pt } }
      }
  }
```

(*End definition for* `\__coffin_get_pole:NnN`*. This function is documented on page* **??**.)

__coffin_reset_structure:N

Resetting the structure is a simple copy job.

```
\cs_new_protected:Npn \__coffin_reset_structure:N #1
  {
    \prop_set_eq:cN { l__coffin_corners_ \__int_value:w #1 _prop }
      \c__coffin_corners_prop
    \prop_set_eq:cN { l__coffin_poles_ \__int_value:w #1 _prop }
      \c__coffin_poles_prop
  }
```

(*End definition for* `\__coffin_reset_structure:N`*. This function is documented on page* **??**.)

__coffin_set_eq_structure:NN
__coffin_gset_eq_structure:NN

Setting coffin structures equal simply means copying the property list.

```
\cs_new_protected:Npn \__coffin_set_eq_structure:NN #1#2
  {
    \prop_set_eq:cc { l__coffin_corners_ \__int_value:w #1 _prop }
      { l__coffin_corners_ \__int_value:w #2 _prop }
    \prop_set_eq:cc { l__coffin_poles_ \__int_value:w #1 _prop }
      { l__coffin_poles_ \__int_value:w #2 _prop }
  }
\cs_new_protected:Npn \__coffin_gset_eq_structure:NN #1#2
  {
```

```
    \prop_gset_eq:cc { l__coffin_corners_ \__int_value:w #1 _prop }
      { l__coffin_corners_ \__int_value:w #2 _prop }
    \prop_gset_eq:cc { l__coffin_poles_ \__int_value:w #1 _prop }
      { l__coffin_poles_ \__int_value:w #2 _prop }
  }
```

(*End definition for* `\__coffin_set_eq_structure:NN` *and* `\__coffin_gset_eq_structure:NN`. *These functions are documented on page* **??**.)

`\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.

```
\cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
  {
    \__coffin_if_exist:NT #1
      {
        \__coffin_set_pole:Nnx #1 {#2}
          {
            { 0 pt } { \dim_eval:n {#3} }
            { 1000 pt } { 0 pt }
          }
      }
  }
\cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
  {
    \__coffin_if_exist:NT #1
      {
        \__coffin_set_pole:Nnx #1 {#2}
          {
            { \dim_eval:n {#3} } { 0 pt }
            { 0 pt } { 1000 pt }
          }
      }
  }
\cs_new_protected:Npn \__coffin_set_pole:Nnn #1#2#3
  { \prop_put:cnn { l__coffin_poles_ \__int_value:w #1 _prop } {#2} {#3} }
\cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
\cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
\cs_generate_variant:Nn \__coffin_set_pole:Nnn { Nnx }
```

(*End definition for* `\coffin_set_horizontal_pole:Nnn` *and* `\coffin_set_horizontal_pole:cnn`. *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.

```
\cs_new_protected:Npn \__coffin_update_corners:N #1
  {
    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { tl }
      { { 0 pt } { \dim_use:N \box_ht:N #1 } }
    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { tr }
      { { \dim_use:N \box_wd:N #1 } { \dim_use:N \box_ht:N #1 } }
```

```
    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { bl }
      { { 0 pt } { \dim_eval:n { - \box_dp:N #1 } } }
    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } { br }
      { { \dim_use:N \box_wd:N #1 } { \dim_eval:n { - \box_dp:N #1 } } }
  }
```

(*End definition for* `\__coffin_update_corners:N`. *This function is documented on page* **??**.)

`\__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.

```
\cs_new_protected:Npn \__coffin_update_poles:N #1
  {
    \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { hc }
      {
        { \dim_eval:n { 0.5 \box_wd:N #1 } }
        { 0 pt } { 0 pt } { 1000 pt }
      }
    \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { r }
      {
        { \dim_use:N \box_wd:N #1 }
        { 0 pt } { 0 pt } { 1000 pt }
      }
    \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { vc }
      {
        { 0 pt }
        { \dim_eval:n { ( \box_ht:N #1 - \box_dp:N #1 ) / 2 } }
        { 1000 pt }
        { 0 pt }
      }
    \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { t }
      {
        { 0 pt }
        { \dim_use:N \box_ht:N #1 }
        { 1000 pt }
        { 0 pt }
      }
    \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } { b }
      {
        { 0 pt }
        { \dim_eval:n { - \box_dp:N #1 } }
        { 1000 pt }
        { 0 pt }
      }
  }
```

(*End definition for* `\__coffin_update_poles:N`. *This function is documented on page* **??**.)

15.5 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.

```
\cs_new_protected:Npn \__coffin_calculate_intersection:Nnn #1#2#3
  {
    \__coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
    \__coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
    \bool_set_false:N \l__coffin_error_bool
    \exp_last_two_unbraced:Noo
      \__coffin_calculate_intersection:nnnnnnnn
        \l__coffin_pole_a_tl \l__coffin_pole_b_tl
    \bool_if:NT \l__coffin_error_bool
      {
        \__msg_kernel_error:nn { kernel } { no-pole-intersection }
        \dim_zero:N \l__coffin_x_dim
        \dim_zero:N \l__coffin_y_dim
      }
  }
```

The two poles passed here each have four values (as dimensions), (a, b, c, d) and (a', b', c', d'). These are arguments 1–4 and 5–8, respectively. In both cases a and b are the 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.

```
\cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnnnn
  #1#2#3#4#5#6#7#8
  {
    \dim_compare:nNnTF {#3} = { \c_zero_dim }
```

The case where the first pole is vertical. So the x-component of the interaction will be at a. There is then a test on the second pole: if it is also vertical then there is an error.

```
      {
        \dim_set:Nn \l__coffin_x_dim {#1}
        \dim_compare:nNnTF {#7} = \c_zero_dim
          { \bool_set_true:N \l__coffin_error_bool }
```

The second pole may still be horizontal, in which case the y-component of the intersection 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.

```
          {
            \dim_compare:nNnTF {#8} = \c_zero_dim
              { \dim_set:Nn \l__coffin_y_dim {#6} }
              {
                \__coffin_calculate_intersection_aux:nnnnnN
```

```
          {#1} {#5} {#6} {#7} {#8} \l__coffin_y_dim
        }
      }
    }
```

If the first pole is not vertical then it may be horizontal. If so, then the procedure is essentially the same as that already done but with the x- and y-components interchanged.

```
    {
     \dim_compare:nNnTF {#4} = \c_zero_dim
      {
       \dim_set:Nn \l__coffin_y_dim {#2}
       \dim_compare:nNnTF {#8} = { \c_zero_dim }
        { \bool_set_true:N \l__coffin_error_bool }
        {
         \dim_compare:nNnTF {#7} = \c_zero_dim
          { \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.

```
          {
           \__coffin_calculate_intersection_aux:nnnnnN
            {#2} {#6} {#5} {#8} {#7} \l__coffin_x_dim
          }
        }
      }
```

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.

```
      {
       \dim_compare:nNnTF {#7} = \c_zero_dim
        {
         \dim_set:Nn \l__coffin_x_dim {#5}
         \__coffin_calculate_intersection_aux:nnnnnN
          {#5} {#1} {#2} {#3} {#4} \l__coffin_y_dim
        }
        {
         \dim_compare:nNnTF {#8} = \c_zero_dim
          {
           \dim_set:Nn \l__coffin_y_dim {#6}
           \__coffin_calculate_intersection_aux:nnnnnN
            {#6} {#2} {#1} {#4} {#3} \l__coffin_x_dim
          }
```

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.

```
          {
```

```
                    \fp_set:Nn \l__coffin_slope_x_fp
                      { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
                    \fp_set:Nn \l__coffin_slope_y_fp
                      { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
                    \fp_compare:nNnTF
                      \l__coffin_slope_x_fp = \l__coffin_slope_y_fp
                      { \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.

```
                      {
                        \dim_set:Nn \l__coffin_x_dim
                          {
                            \fp_to_dim:n
                              {
                                (
                                      \dim_to_fp:n {#1} * \l__coffin_slope_x_fp
                                  - ( \dim_to_fp:n {#5} * \l__coffin_slope_y_fp )
                                  -   \dim_to_fp:n {#2}
                                  +   \dim_to_fp:n {#6}
                                )
                                /
                                ( \l__coffin_slope_x_fp - \l__coffin_slope_y_fp )
                            }
                          }
                        \__coffin_calculate_intersection_aux:nnnnnN
                          { \l__coffin_x_dim }
                          {#5} {#6} {#8} {#7} \l__coffin_y_dim
                      }
                  }
              }
          }
      }
  }
```

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.

```
\cs_new_protected:Npn \__coffin_calculate_intersection_aux:nnnnnN #1#2#3#4#5#6
  {
    \dim_set:Nn #6
```

```
      {
        \fp_to_dim:n
          {
            \dim_to_fp:n {#4} *
            ( \dim_to_fp:n {#1} - \dim_to_fp:n {#2} ) /
            \dim_to_fp:n {#5}
            + \dim_to_fp:n {#3}
          }
      }
  }
```

(*End definition for* `\__coffin_calculate_intersection:Nnn`. *This function is documented on page* **??**.)

15.6 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.

```
\cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
  {
    \__coffin_align:NnnNnnnnN
      #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
```

Correct the placement of the reference point. If the x-offset is negative then the reference point of the second box is to the left of that of the first, which is corrected using a kern. On the right side the first box might stick out, which 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.

```
    \hbox_set:Nn \l__coffin_aligned_coffin
      {
        \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
          { \tex_kern:D -\l__coffin_offset_x_dim }
        \hbox_unpack:N \l__coffin_aligned_coffin
        \dim_set:Nn \l__coffin_internal_dim
          { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
        \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
          { \tex_kern:D -\l__coffin_internal_dim }
      }
```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```
   \__coffin_reset_structure:N \l__coffin_aligned_coffin
    \prop_clear:c
      {  l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _ prop }
    \__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.

```
    \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
```

```
      {
        \__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { 0 pt }
        \__coffin_offset_poles:Nnn #4 { 0 pt } { \l__coffin_offset_y_dim }
        \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { 0 pt }
        \__coffin_offset_corners:Nnn #4 { 0 pt } { \l__coffin_offset_y_dim }
      }
      {
        \__coffin_offset_poles:Nnn #1 { 0 pt } { 0 pt }
        \__coffin_offset_poles:Nnn #4
          { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
        \__coffin_offset_corners:Nnn #1 { 0 pt } { 0 pt }
        \__coffin_offset_corners:Nnn #4
          { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
      }
    \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
    \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
  }
\cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cnnc }
```

(*End definition for* `\coffin_join:NnnNnnnn` *and others. These functions are documented on page* **??**.)

`\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.

```
\cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
  {
    \__coffin_align:NnnNnnnnN
      #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
    \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
    \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
    \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
    \__coffin_reset_structure:N \l__coffin_aligned_coffin
    \prop_set_eq:cc
      { l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _prop }
      { l__coffin_corners_ \__int_value:w #1 _prop }
    \__coffin_update_poles:N  \l__coffin_aligned_coffin
    \__coffin_offset_poles:Nnn #1 { 0 pt } { 0 pt }
    \__coffin_offset_poles:Nnn #4
      { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
    \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
    \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
  }
\cs_new_protected:Npn \coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
  {
    \__coffin_align:NnnNnnnnN
      #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
    \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
    \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
    \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
```

```
    \box_set_eq:NN #1 \l__coffin_aligned_coffin
  }
\cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnnc }
```

(*End definition for* `\coffin_attach:NnnNnnnn` *and others. These functions are documented on page* **??**.)

`\__coffin_align:NnnNnnnnN` The internal function aligns the two coffins into a third one, but performs no corrections on the resulting coffin poles. The process begins by finding the points of intersection for the poles for each of the input coffins. Those for the first coffin are worked out after those for the second coffin, as this allows the 'primed' storage area to be used for the second coffin. The 'real' box offsets are then calculated, before using these to re-box the input coffins. The default poles are then set up, but the final result will depend on how the bounding box is being handled.

```
\cs_new_protected:Npn \__coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
  {
    \__coffin_calculate_intersection:Nnn #4 {#5} {#6}
    \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
    \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
    \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
    \dim_set:Nn \l__coffin_offset_x_dim
      { \l__coffin_x_dim - \l__coffin_x_prime_dim + #7 }
    \dim_set:Nn \l__coffin_offset_y_dim
      { \l__coffin_y_dim - \l__coffin_y_prime_dim + #8 }
    \hbox_set:Nn \l__coffin_aligned_internal_coffin
      {
        \box_use:N #1
        \tex_kern:D -\box_wd:N #1
        \tex_kern:D \l__coffin_offset_x_dim
        \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #4 }
      }
    \coffin_set_eq:NN #9 \l__coffin_aligned_internal_coffin
  }
```

(*End definition for* `\__coffin_align:NnnNnnnnN`*. This function is documented on page* **??**.)

`\__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.

```
\cs_new_protected:Npn \__coffin_offset_poles:Nnn #1#2#3
  {
    \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
      { \__coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {#2} {#3} }
  }
\cs_new_protected:Npn \__coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
  {
    \dim_set:Nn \l__coffin_x_dim { #3 + #7 }
```

```
    \dim_set:Nn \l__coffin_y_dim { #4 + #8 }
    \tl_if_in:nnTF {#2} { - }
      { \tl_set:Nn \l__coffin_internal_tl { {#2} } }
      { \tl_set:Nn \l__coffin_internal_tl { { #1 - #2 } } }
    \exp_last_unbraced:NNo \__coffin_set_pole:Nnx \l__coffin_aligned_coffin
      { \l__coffin_internal_tl }
      {
        { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
        {#5} {#6}
      }
  }
```

(*End definition for* `\__coffin_offset_poles:Nnn`. *This function is documented on page* **??**.)

`\__coffin_offset_corners:Nnn`
`\__coffin_offset_corner:Nnnnn`

Saving the offset corners of a coffin is very similar, except that there is no need to worry about naming: every corner can be saved here as order is unimportant.

```
\cs_new_protected:Npn \__coffin_offset_corners:Nnn #1#2#3
  {
    \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
      { \__coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
  }
\cs_new_protected:Npn \__coffin_offset_corner:Nnnnn #1#2#3#4#5#6
  {
    \prop_put:cnx
      { l__coffin_corners_ \__int_value:w \l__coffin_aligned_coffin _prop }
      { #1 - #2 }
      {
        { \dim_eval:n { #3 + #5 } }
        { \dim_eval:n { #4 + #6 } }
      }
  }
```

(*End definition for* `\__coffin_offset_corners:Nnn`. *This function is documented on page* **??**.)

`\__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.

```
\cs_new_protected:Npn \__coffin_update_vertical_poles:NNN #1#2#3
  {
    \__coffin_get_pole:NnN #3 { #1 -T } \l__coffin_pole_a_tl
    \__coffin_get_pole:NnN #3 { #2 -T } \l__coffin_pole_b_tl
    \exp_last_two_unbraced:Noo \__coffin_update_T:nnnnnnnnN
      \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
    \__coffin_get_pole:NnN #3 { #1 -B } \l__coffin_pole_a_tl
    \__coffin_get_pole:NnN #3 { #2 -B } \l__coffin_pole_b_tl
    \exp_last_two_unbraced:Noo \__coffin_update_B:nnnnnnnnN
      \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
  }
\cs_new_protected:Npn \__coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
  {
    \dim_compare:nNnTF {#2} < {#6}
```

```
      {
        \__coffin_set_pole:Nnx #9 { T }
          { { 0 pt } {#6} { 1000 pt } { 0 pt } }
      }
      {
        \__coffin_set_pole:Nnx #9 { T }
          { { 0 pt } {#2} { 1000 pt } { 0 pt } }
      }
  }
\cs_new_protected:Npn \__coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
  {
    \dim_compare:nNnTF {#2} < {#6}
      {
        \__coffin_set_pole:Nnx #9 { B }
          { { 0 pt } {#2}  { 1000 pt } { 0 pt } }
      }
      {
        \__coffin_set_pole:Nnx #9 { B }
          { { 0 pt } {#6} { 1000 pt } { 0 pt } }
      }
  }
```

(*End definition for* `\__coffin_update_vertical_poles:NNN`*. This function is documented on page* **??**.)

`\coffin_typeset:Nnnnn`
`\coffin_typeset:cnnnn`

Typesetting a coffin means aligning it with the current position, which is done using a coffin with no content at all. As well as aligning to the empty coffin, there is also a need to leave vertical mode, if necessary.

```
\cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
  {
    \hbox_unpack:N \c_empty_box
    \__coffin_align:NnnNnnnnN \c_empty_coffin { H } { l }
      #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
    \box_use:N \l__coffin_aligned_coffin
  }
\cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }
```

(*End definition for* `\coffin_typeset:Nnnnn` *and* `\coffin_typeset:cnnnn`*. These functions are documented on page* **??**.)

15.7 Coffin diagnostics

`\l__coffin_display_coffin`
`\l__coffin_display_coord_coffin`
`\l__coffin_display_pole_coffin`

Used for printing coffins with data structures attached.

```
\coffin_new:N \l__coffin_display_coffin
\coffin_new:N \l__coffin_display_coord_coffin
\coffin_new:N \l__coffin_display_pole_coffin
```

(*End definition for* `\l__coffin_display_coffin`*. This variable is documented on page* **??**.)

`\l__coffin_display_handles_prop`

This property list is used to print coffin handles at suitable positions. The offsets are expressed as multiples of the basic offset value, which therefore acts as a scale-factor.

```
\prop_new:N \l__coffin_display_handles_prop
```

```
\prop_put:Nnn \l__coffin_display_handles_prop { tl }
  { { b } { r } { -1 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { thc }
  { { b } { hc } { 0 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { tr }
  { { b } { l } { 1 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { vcl }
  { { vc } { r } { -1 } { 0 } }
\prop_put:Nnn \l__coffin_display_handles_prop { vchc }
  { { vc } { hc } { 0 } { 0 } }
\prop_put:Nnn \l__coffin_display_handles_prop { vcr }
  { { vc } { l } { 1 } { 0 } }
\prop_put:Nnn \l__coffin_display_handles_prop { bl }
  { { t } { r } { -1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { bhc }
  { { t } { hc } { 0 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { br }
  { { t } { l } { 1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Tl }
  { { t } { r } { -1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Thc }
  { { t } { hc } { 0 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Tr }
  { { t } { l } { 1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Hl }
  { { vc } { r } { -1 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
  { { vc } { hc } { 0 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Hr }
  { { vc } { l } { 1 } { 1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Bl }
  { { b } { r } { -1 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Bhc }
  { { b } { hc } { 0 } { -1 } }
\prop_put:Nnn \l__coffin_display_handles_prop { Br }
  { { b } { l } { 1 } { -1 } }
```

(*End definition for* `\l__coffin_display_handles_prop`. *This variable is documented on page* **??**.)

`\l__coffin_display_offset_dim`

The standard offset for the label from the handle position when displaying handles.

```
\dim_new:N  \l__coffin_display_offset_dim
\dim_set:Nn \l__coffin_display_offset_dim { 2 pt }
```

(*End definition for* `\l__coffin_display_offset_dim`. *This variable is documented on page* **??**.)

`\l__coffin_display_x_dim`
`\l__coffin_display_y_dim`

As the intersections of poles have to be calculated to find which ones to print, there is a need to avoid repetition. This is done by saving the intersection into two dedicated values.

```
\dim_new:N \l__coffin_display_x_dim
\dim_new:N \l__coffin_display_y_dim
```

(*End definition for* `\l__coffin_display_x_dim`. *This variable is documented on page* **??**.)

\l__coffin_display_poles_prop

A property list for printing poles: various things need to be deleted from this to get a "nice" output.

```
\prop_new:N \l__coffin_display_poles_prop
```

(*End definition for* `\l__coffin_display_poles_prop`*. This variable is documented on page* ??.)

\l__coffin_display_font_tl

Stores the settings used to print coffin data: this keeps things flexible.

```
\tl_new:N  \l__coffin_display_font_tl
⟨*initex⟩
\tl_set:Nn \l__coffin_display_font_tl { } % TODO
⟨/initex⟩
⟨*package⟩
\tl_set:Nn \l__coffin_display_font_tl { \sffamily \tiny }
⟨/package⟩
```

(*End definition for* `\l__coffin_display_font_tl`*. This variable is documented on page* ??.)

\coffin_mark_handle:Nnnn
\coffin_mark_handle:cnnn
__coffin_mark_handle_aux:nnnnNnn

Marking a single handle is relatively easy. The standard attachment function is used, meaning that there are two calculations for the location. However, this is likely to be okay given the load expected. Contrast with the more optimised version for showing all handles which comes next.

```
\cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
  {
    \hcoffin_set:Nn \l__coffin_display_pole_coffin
      {
⟨*initex⟩
        \hbox:n { \tex_vrule:D width 1 pt height 1 pt \scan_stop: } % TODO
⟨/initex⟩
⟨*package⟩
        \color {#4}
        \rule { 1 pt } { 1 pt }
⟨/package⟩
      }
    \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
      \l__coffin_display_pole_coffin { hc } { vc } { 0 pt } { 0 pt }
    \hcoffin_set:Nn \l__coffin_display_coord_coffin
      {
⟨*initex⟩
        % TODO
⟨/initex⟩
⟨*package⟩
        \color {#4}
⟨/package⟩
        \l__coffin_display_font_tl
        ( \tl_to_str:n { #2 , #3 } )
      }
    \prop_get:NnN \l__coffin_display_handles_prop
      { #2 #3 } \l__coffin_internal_tl
    \quark_if_no_value:NTF \l__coffin_internal_tl
      {
        \prop_get:NnN \l__coffin_display_handles_prop
```

```
        { #3 #2 } \l__coffin_internal_tl
      \quark_if_no_value:NTF \l__coffin_internal_tl
        {
          \coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
            \l__coffin_display_coord_coffin { l } { vc }
              { 1 pt } { 0 pt }
        }
        {
          \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
            \l__coffin_internal_tl #1 {#2} {#3}
        }
    }
    {
      \exp_last_unbraced:No \__coffin_mark_handle_aux:nnnnNnn
        \l__coffin_internal_tl #1 {#2} {#3}
    }
  }
\cs_new_protected:Npn \__coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
  {
    \coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
      \l__coffin_display_coord_coffin {#1} {#2}
      { #3 \l__coffin_display_offset_dim }
      { #4 \l__coffin_display_offset_dim }
  }
\cs_generate_variant:Nn \coffin_mark_handle:Nnnn { c }
```

(*End definition for* `\coffin_mark_handle:Nnnn` *and* `\coffin_mark_handle:cnnn`*. These functions are documented on page* **??**.)

`\coffin_display_handles:Nn`
`\coffin_display_handles:cn`
`\__coffin_display_handles_aux:nnnnnn`
`\__coffin_display_handles_aux:nnnn`
`\__coffin_display_attach:Nnnnn`

Printing the poles starts by removing any duplicates, for which the `H` poles is used as the definitive version for the baseline and bottom. Two loops are then used to find the combinations of handles for all of these poles. This is done such that poles are removed during the loops to avoid duplication.

```
\cs_new_protected:Npn \coffin_display_handles:Nn #1#2
  {
    \hcoffin_set:Nn \l__coffin_display_pole_coffin
      {
⟨*initex⟩
        \hbox:n { \tex_vrule:D width 1 pt height 1 pt \scan_stop: } % TODO
⟨/initex⟩
⟨*package⟩
        \color {#2}
        \rule { 1 pt } { 1 pt }
⟨/package⟩
      }
    \prop_set_eq:Nc \l__coffin_display_poles_prop
      { l__coffin_poles_ \__int_value:w #1 _prop }
    \__coffin_get_pole:NnN #1 { H } \l__coffin_pole_a_tl
    \__coffin_get_pole:NnN #1 { T } \l__coffin_pole_b_tl
    \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
```

```
      { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
    \__coffin_get_pole:NnN #1 { B } \l__coffin_pole_b_tl
    \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
      { \prop_remove:Nn \l__coffin_display_poles_prop { B } }
    \coffin_set_eq:NN \l__coffin_display_coffin #1
    \prop_map_inline:Nn \l__coffin_display_poles_prop
      {
        \prop_remove:Nn \l__coffin_display_poles_prop {##1}
        \__coffin_display_handles_aux:nnnnnn {##1} ##2 {#2}
      }
    \box_use:N \l__coffin_display_coffin
  }
```

For each pole there is a check for an intersection, which here does not give an error if none is found. The successful values are stored and used to align the pole coffin with the main coffin for output. The positions are recovered from the preset list if available.

```
\cs_new_protected:Npn \__coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
  {
    \prop_map_inline:Nn \l__coffin_display_poles_prop
      {
        \bool_set_false:N \l__coffin_error_bool
        \__coffin_calculate_intersection:nnnnnnnn {#2} {#3} {#4} {#5} ##2
        \bool_if:NF \l__coffin_error_bool
          {
            \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
            \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
            \__coffin_display_attach:Nnnnn
              \l__coffin_display_pole_coffin { hc } { vc }
              { 0 pt } { 0 pt }
            \hcoffin_set:Nn \l__coffin_display_coord_coffin
              {
⟨*initex⟩
                % TODO
⟨/initex⟩
⟨*package⟩
                \color {#6}
⟨/package⟩
                \l__coffin_display_font_tl
                ( \tl_to_str:n { #1 , ##1 } )
              }
            \prop_get:NnN \l__coffin_display_handles_prop
              { #1 ##1 } \l__coffin_internal_tl
            \quark_if_no_value:NTF \l__coffin_internal_tl
              {
                \prop_get:NnN \l__coffin_display_handles_prop
                  { ##1 #1 } \l__coffin_internal_tl
                \quark_if_no_value:NTF \l__coffin_internal_tl
                  {
                    \__coffin_display_attach:Nnnnn
                      \l__coffin_display_coord_coffin { l } { vc }
```

```
                      { 1 pt } { 0 pt }
                  }
                  {
                    \exp_last_unbraced:No
                      \__coffin_display_handles_aux:nnnn
                      \l__coffin_internal_tl
                  }
              }
              {
                \exp_last_unbraced:No \__coffin_display_handles_aux:nnnn
                  \l__coffin_internal_tl
              }
          }
      }
  }
\cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
  {
    \__coffin_display_attach:Nnnnn
      \l__coffin_display_coord_coffin {#1} {#2}
      { #3 \l__coffin_display_offset_dim }
      { #4 \l__coffin_display_offset_dim }
  }
\cs_generate_variant:Nn \coffin_display_handles:Nn { c }
```

This is a dedicated version of `\coffin_attach:NnnNnnnn` with a hard-wired first coffin. As the intersection is already known and stored for the display coffin the code simply uses it directly, with no calculation.

```
\cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5
  {
    \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
    \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
    \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
    \dim_set:Nn \l__coffin_offset_x_dim
      { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
    \dim_set:Nn \l__coffin_offset_y_dim
      { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
    \hbox_set:Nn \l__coffin_aligned_coffin
      {
        \box_use:N \l__coffin_display_coffin
        \tex_kern:D -\box_wd:N \l__coffin_display_coffin
        \tex_kern:D \l__coffin_offset_x_dim
        \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
      }
    \box_set_ht:Nn \l__coffin_aligned_coffin
      { \box_ht:N \l__coffin_display_coffin }
    \box_set_dp:Nn \l__coffin_aligned_coffin
      { \box_dp:N \l__coffin_display_coffin }
    \box_set_wd:Nn \l__coffin_aligned_coffin
      { \box_wd:N \l__coffin_display_coffin }
    \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
```

```
  }
```

(*End definition for* `\coffin_display_handles:Nn` *and* `\coffin_display_handles:cn`*. These functions are documented on page* ??.)

`\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.

```
\cs_new_protected:Npn \coffin_show_structure:N #1
  {
    \__coffin_if_exist:NT #1
      {
        \__msg_show_variable:Nnn #1 { coffins }
          {
            \prop_map_function:cN
              { l__coffin_poles_ \__int_value:w #1 _prop }
              \__msg_show_item_unbraced:nn
          }
      }
  }
\cs_generate_variant:Nn \coffin_show_structure:N { c }
```

(*End definition for* `\coffin_show_structure:N` *and* `\coffin_show_structure:c`*. These functions are documented on page* ??.)

15.8 Messages

```
\__msg_kernel_new:nnnn { kernel } { no-pole-intersection }
  { No~intersection~between~coffin~poles. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~was~asked~to~find~the~intersection~between~two~poles,~
    but~they~do~not~have~a~unique~meeting~point:~
    the~value~(0~pt,~0~pt)~will~be~used.
  }
\__msg_kernel_new:nnnn { kernel } { unknown-coffin }
  { Unknown~coffin~'#1'. }
  { The~coffin~'#1'~was~never~defined. }
\__msg_kernel_new:nnnn { kernel } { unknown-coffin-pole }
  { Pole~'#1'~unknown~for~coffin~'#2'. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~was~asked~to~find~a~typesetting~pole~for~a~coffin,~
    but~either~the~coffin~does~not~exist~or~the~pole~name~is~wrong.
  }
\__msg_kernel_new:nnn { kernel } { show-coffins }
  {
    Size~of~coffin~\token_to_str:N #1 : \\
    > ~ ht~=~\dim_use:N \box_ht:N #1 \\
    > ~ dp~=~\dim_use:N \box_dp:N #1 \\
    > ~ wd~=~\dim_use:N \box_wd:N #1 \\
    Poles~of~coffin~\token_to_str:N #1 :
```

```
  }
⟨/initex | package⟩
```

16 l3color Implementation

```
⟨*initex | package⟩
```

\color_group_begin:
\color_group_end:

Grouping for color is almost the same as using the basic \group_begin: and \group_end: functions. However, in vertical mode the end-of-group needs a \par, which in horizontal mode does nothing.

```
\cs_new_eq:NN \color_group_begin: \group_begin:
\cs_new_protected_nopar:Npn \color_group_end:
  {
      \tex_par:D
    \group_end:
  }
```

(*End definition for* \color_group_begin: *and* \color_group_end:*. These functions are documented on page 140.*)

\color_ensure_current:

A driver-independent wrapper for setting the foreground color to the current color "now".

```
⟨*initex⟩
\cs_new_protected_nopar:Npn \color_ensure_current:
  { \__driver_color_ensure_current: }
⟨/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.

```
⟨*package⟩
\cs_new_protected_nopar:Npn \color_ensure_current: { }
\AtBeginDocument
  {
    \cs_if_exist:NTF \__driver_color_ensure_current:
      {
        \cs_set_protected_nopar:Npn \color_ensure_current:
          { \__driver_color_ensure_current: }
      }
      {
        \cs_if_exist:NT \set@color
          { \cs_set_protected_nopar:Npn \color_ensure_current: { \set@color } }
      }
  }
⟨/package⟩
```

(*End definition for* \color_ensure_current:*. This function is documented on page 140.*)

```
⟨/initex | package⟩
```

17 l3msg implementation

```
⟨*initex | package⟩
⟨@@=msg⟩
```

\l__msg_internal_tl

A general scratch for the module.

```
\tl_new:N \l__msg_internal_tl
```

(*End definition for* \l__msg_internal_tl. *This variable is documented on page* ??.)

17.1 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
\c__msg_more_text_prefix_tl

Locations for the text of messages.

```
\tl_const:Nn \c__msg_text_prefix_tl      { msg~text~>~ }
\tl_const:Nn \c__msg_more_text_prefix_tl { msg~extra~text~>~ }
```

(*End definition for* \c__msg_text_prefix_tl *and* \c__msg_more_text_prefix_tl. *These variables are documented on page* ??.)

\msg_if_exist_p:nn
\msg_if_exist:nn*TF*

Test whether the control sequence containing the message text exists or not.

```
\prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
  {
    \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
      { \prg_return_true: } { \prg_return_false: }
  }
```

(*End definition for* \msg_if_exist:nn. *These functions are documented on page* 142.)

__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.

```
\cs_new_protected:Npn \__chk_if_free_msg:nn #1#2
  {
    \msg_if_exist:nnT {#1} {#2}
      {
        \__msg_kernel_error:nnxx { kernel } { message-already-defined }
          {#1} {#2}
      }
  }
⟨*package⟩
\tex_ifodd:D \l@expl@log@functions@bool
  \cs_gset_protected:Npn \__chk_if_free_msg:nn #1#2
    {
      \msg_if_exist:nnT {#1} {#2}
        {
          \__msg_kernel_error:nnxx { kernel } { message-already-defined }
```

```
              {#1} {#2}
          }
        \iow_log:x { Defining~message~ #1 / #2 ~\msg_line_context: }
      }
\fi:
⟨/package⟩
```

(*End definition for* `\__chk_if_free_msg:nn`.)

`\msg_new:nnnn`
`\msg_new:nnn`
`\msg_gset:nnnn`
`\msg_gset:nnn`
`\msg_set:nnnn`
`\msg_set:nnn`

Setting a message simply means saving the appropriate text into two functions. A sanity check first.

```
\cs_new_protected:Npn \msg_new:nnnn #1#2
  {
    \__chk_if_free_msg:nn {#1} {#2}
    \msg_gset:nnnn {#1} {#2}
  }
\cs_new_protected:Npn \msg_new:nnn #1#2#3
  { \msg_new:nnnn {#1} {#2} {#3} { } }
\cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
  {
    \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
      ##1##2##3##4 {#3}
    \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
      ##1##2##3##4 {#4}
  }
\cs_new_protected:Npn \msg_set:nnn #1#2#3
  { \msg_set:nnnn {#1} {#2} {#3} { } }
\cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
  {
    \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
      ##1##2##3##4 {#3}
    \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
      ##1##2##3##4 {#4}
  }
\cs_new_protected:Npn \msg_gset:nnn #1#2#3
  { \msg_gset:nnnn {#1} {#2} {#3} { } }
```

(*End definition for* `\msg_new:nnnn` *and* `\msg_new:nnn`*. These functions are documented on page* **??**.)

17.2 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.

```
\tl_const:Nn \c_msg_coding_error_text_tl
  {
    This~is~a~coding~error.
    \\ \\
  }
\tl_const:Nn \c_msg_continue_text_tl
  { Type~<return>~to~continue }
\tl_const:Nn \c_msg_critical_text_tl
  { Reading~the~current~file~will~stop. }
```

```
\tl_const:Nn \c_msg_fatal_text_tl
  { This~is~a~fatal~error:~LaTeX~will~abort. }
\tl_const:Nn \c_msg_help_text_tl
  { For~immediate~help~type~H~<return> }
\tl_const:Nn \c_msg_no_info_text_tl
  {
    LaTeX~does~not~know~anything~more~about~this~error,~sorry.
    \c_msg_return_text_tl
  }
\tl_const:Nn \c_msg_on_line_text_tl { on~line }
\tl_const:Nn \c_msg_return_text_tl
  {
    \\ \\
    Try~typing~<return>~to~proceed.
    \\
    If~that~doesn't~work,~type~X~<return>~to~quit.
  }
\tl_const:Nn \c_msg_trouble_text_tl
  {
    \\ \\
    More~errors~will~almost~certainly~follow: \\
    the~LaTeX~run~should~be~aborted.
  }
```

(*End definition for* `\c_msg_coding_error_text_tl` *and others. These variables are documented on page* ??.)

`\msg_line_number:`
`\msg_line_context:`

For writing the line number nicely. `\msg_line_context:` was set up earlier, so this is not `new`.

```
\cs_new_nopar:Npn \msg_line_number: { \int_use:N \tex_inputlineno:D }
\cs_gset_nopar:Npn \msg_line_context:
  {
    \c_msg_on_line_text_tl
    \c_space_tl
    \msg_line_number:
  }
```

(*End definition for* `\msg_line_number:` *and* `\msg_line_context:`*. These functions are documented on page 142.*)

17.3 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`.

```
\cs_new_protected:Npn \msg_interrupt:nnn #1#2#3
  {
    \tl_if_empty:nTF {#3}
      {
        \__msg_interrupt_wrap:nn { \\ \c_msg_no_info_text_tl }
```

```
        {#1 \\\\ #2 \\\\ \c_msg_continue_text_tl }
    }
    {
      \__msg_interrupt_wrap:nn { \\ #3 }
        {#1 \\\\ #2 \\\\ \c_msg_help_text_tl }
    }
}
```

(*End definition for* `\msg_interrupt:nnn`*. This function is documented on page* *146*.)

`\__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.

```
\cs_new_protected:Npn \__msg_interrupt_wrap:nn #1#2
  {
    \iow_wrap:nnnN {#1} { | ~ } { } \__msg_interrupt_more_text:n
    \iow_wrap:nnnN {#2} { ! ~ } { } \__msg_interrupt_text:n
  }
\cs_new_protected:Npn \__msg_interrupt_more_text:n #1
  {
    \exp_args:Nx \tex_errhelp:D
      {
        |''''''''''''''''''''''''''''''''''''''''''''''''
        #1 \iow_newline:
        |................................................
      }
  }
```

(*End definition for* `\__msg_interrupt_wrap:nn`*.*)

`\__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 bee inserted after the message is entirely cleaned up.

```
\group_begin:
  \char_set_lccode:nn {`\{} {`\ }
  \char_set_lccode:nn {`\}} {`\ }
  \char_set_lccode:nn {`\&} {`\!}
  \char_set_catcode_active:N \&
\tl_to_lowercase:n
  {
    \group_end:
    \cs_new_protected:Npn \__msg_interrupt_text:n #1
```

```
      {
        \iow_term:x
          {
            \iow_newline:
            !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            \iow_newline:
            !
          }
        \group_begin:
          \cs_set_protected_nopar:Npn &
            {
              \tex_errmessage:D
                {
                  #1
                  \use_none:n
                    { ............................................ }
                }
            }
          \exp_after:wN
        \group_end:
        &
      }
  }
```

(*End definition for* `\__msg_interrupt_text:n`*.*)

`\msg_log:n`
`\msg_term:n`

Printing to the log or terminal without a stop is rather easier. A bit of simple visual work sets things off nicely.

```
\cs_new_protected:Npn \msg_log:n #1
  {
    \iow_log:n { ................................................. }
    \iow_wrap:nnnN { . ~ #1} { . ~ } { } \iow_log:n
    \iow_log:n { ................................................. }
  }
\cs_new_protected:Npn \msg_term:n #1
  {
    \iow_term:n { ************************************************* }
    \iow_wrap:nnnN { * ~ #1} { * ~ } { } \iow_term:n
    \iow_term:n { ************************************************* }
  }
```

(*End definition for* `\msg_log:n`*. This function is documented on page* *147**.*)

17.4 Displaying messages

LaTeX is handling error messages and so the TeX ones are disabled. This is already done by the LaTeX 2ε kernel, so to avoid messing up any deliberate change by a user this is only set in format mode.

```
⟨*initex⟩
\int_gset_eq:NN \tex_errorcontextlines:D \c_minus_one
⟨/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.

```
\cs_new:Npn \msg_fatal_text:n #1 { Fatal~#1~error }
\cs_new:Npn \msg_critical_text:n #1 { Critical~#1~error }
\cs_new:Npn \msg_error_text:n #1 { #1~error }
\cs_new:Npn \msg_warning_text:n #1 { #1~warning }
\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 143.*)

\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".

```
\cs_new:Npn \msg_see_documentation_text:n #1
  {
    \\ \\ See~the~
    \str_if_eq:nnTF {#1} { LaTeX } { LaTeX3 } {#1} ~
    documentation~for~further~information.
  }
```

(*End definition for* `\msg_see_documentation_text:n`*. This function is documented on page 143.*)

__msg_class_new:nn

```
\group_begin:
  \cs_set_protected:Npn \__msg_class_new:nn #1#2
    {
      \prop_new:c { l__msg_redirect_ #1 _prop }
      \cs_new_protected:cpn { __msg_ #1 _code:nnnnnn } ##1##2##3##4##5##6 {#2}
      \cs_new_protected:cpn { msg_ #1 :nnnnnn } ##1##2##3##4##5##6
        {
          \use:x
            {
              \exp_not:n { \__msg_use:nnnnnnn {#1} {##1} {##2} }
                { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
                { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
            }
        }
      \cs_new_protected:cpx { msg_ #1 :nnnnn } ##1##2##3##4##5
        { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
      \cs_new_protected:cpx { msg_ #1 :nnnn } ##1##2##3##4
        { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
      \cs_new_protected:cpx { msg_ #1 :nnn } ##1##2##3
        { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
      \cs_new_protected:cpx { msg_ #1 :nn } ##1##2
        { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
      \cs_new_protected:cpx { msg_ #1 :nnxxxx } ##1##2##3##4##5##6
        {
          \use:x
            {
              \exp_not:N \exp_not:n
                { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} }
                {##3} {##4} {##5} {##6}
            }
```

```
      }
    \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
      { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
    \cs_new_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
      { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
    \cs_new_protected:cpx { msg_ #1 :nnx } ##1##2##3
      { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
  }
```

(*End definition for* `\__msg_class_new:nn`*. This function is documented on page* **??**.)

\msg_fatal:nnnnnn
\msg_fatal:nnnnn
\msg_fatal:nnnn
\msg_fatal:nnn
\msg_fatal:nn
\msg_fatal:nnxxxx
\msg_fatal:nnxxx
\msg_fatal:nnxx
\msg_fatal:nnx

For fatal errors, after the error message TEX bails out.

```
\__msg_class_new:nn { fatal }
  {
    \msg_interrupt:nnn
      { \msg_fatal_text:n {#1} : ~ "#2" }
      {
        \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
        \msg_see_documentation_text:n {#1}
      }
      { \c_msg_fatal_text_tl }
    \tex_end:D
  }
```

(*End definition for* `\msg_fatal:nnnnnn` *and others. These functions are documented on page* **??**.)

\msg_critical:nnnnnn
\msg_critical:nnnnn
\msg_critical:nnnn
\msg_critical:nnn
\msg_critical:nn
\msg_critical:nnxxxx
\msg_critical:nnxxx
\msg_critical:nnxx
\msg_critical:nnx

Not quite so bad: just end the current file.

```
\__msg_class_new:nn { critical }
  {
    \msg_interrupt:nnn
      { \msg_critical_text:n {#1} : ~ "#2" }
      {
        \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
        \msg_see_documentation_text:n {#1}
      }
      { \c_msg_critical_text_tl }
    \tex_endinput:D
  }
```

(*End definition for* `\msg_critical:nnnnnn` *and others. These functions are documented on page* **??**.)

\msg_error:nnnnnn
\msg_error:nnnnn
\msg_error:nnnn
\msg_error:nnn
\msg_error:nn
\msg_error:nnxxxx
\msg_error:nnxxx
\msg_error:nnxx
\msg_error:nnx
__msg_error:cnnnnn
__msg_no_more_text:nnnn

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.

```
\__msg_class_new:nn { error }
  {
    \__msg_error:cnnnnn
      { \c__msg_more_text_prefix_tl #1 / #2 }
      {#3} {#4} {#5} {#6}
      {
        \msg_interrupt:nnn
          { \msg_error_text:n {#1} : ~ "#2" }
          {
```

```
          \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
          \msg_see_documentation_text:n {#1}
        }
    }
  }
\cs_new_protected:Npn \__msg_error:cnnnnn #1#2#3#4#5#6
  {
    \cs_if_eq:cNTF {#1} \__msg_no_more_text:nnnn
      { #6 { } }
      { #6 { \use:c {#1} {#2} {#3} {#4} {#5} } }
  }
\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* **??**.)

`\msg_warning:nnnnnn`
`\msg_warning:nnnnn`
`\msg_warning:nnnn`
`\msg_warning:nnn`
`\msg_warning:nn`
`\msg_warning:nnxxxx`
`\msg_warning:nnxxx`
`\msg_warning:nnxx`
`\msg_warning:nnx`

Warnings are printed to the terminal.

```
\__msg_class_new:nn { warning }
  {
    \msg_term:n
      {
        \msg_warning_text:n {#1} : ~ "#2" \\ \\
        \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
      }
  }
```

(*End definition for* `\msg_warning:nnnnnn` *and others. These functions are documented on page* **??**.)

`\msg_info:nnnnnn`
`\msg_info:nnnnn`
`\msg_info:nnnn`
`\msg_info:nnn`
`\msg_info:nn`
`\msg_info:nnxxxx`
`\msg_info:nnxxx`
`\msg_info:nnxx`
`\msg_info:nnx`

Information only goes into the log.

```
\__msg_class_new:nn { info }
  {
    \msg_log:n
      {
        \msg_info_text:n {#1} : ~ "#2" \\ \\
        \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
      }
  }
```

(*End definition for* `\msg_info:nnnnnn` *and others. These functions are documented on page* **??**.)

`\msg_log:nnnnnn`
`\msg_log:nnnnn`
`\msg_log:nnnn`
`\msg_log:nnn`
`\msg_log:nn`
`\msg_log:nnxxxx`
`\msg_log:nnxxx`
`\msg_log:nnxx`
`\msg_log:nnx`

"Log" data is very similar to information, but with no extras added.

```
\__msg_class_new:nn { log }
  {
    \iow_wrap:nnnN
      { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
      { } { } \iow_log:n
  }
```

(*End definition for* `\msg_log:nnnnnn` *and others. These functions are documented on page* **??**.)

`\msg_none:nnnnnn`
`\msg_none:nnnnn`
`\msg_none:nnnn`
`\msg_none:nnn`
`\msg_none:nn`
`\msg_none:nnxxxx`
`\msg_none:nnxxx`
`\msg_none:nnxx`
`\msg_none:nnx`

The `none` message type is needed so that input can be gobbled.

```
\__msg_class_new:nn { none } { }
```

(*End definition for* `\msg_none:nnnnnn` *and others. These functions are documented on page* **??**.)

End the group to eliminate `\__msg_class_new:nn`.

```
\group_end:
```

`\__msg_class_chk_exist:nT`

Checking that a message class exists. We build this from `\cs_if_free:cTF` rather than `\cs_if_exist:cTF` because that avoids reading the second argument earlier than necessary.

```
\cs_new:Npn \__msg_class_chk_exist:nT #1
  {
    \cs_if_free:cTF { __msg_ #1 _code:nnnnnn }
      { \__msg_kernel_error:nnx { kernel } { message-class-unknown } {#1} }
  }
```

(*End definition for* `\__msg_class_chk_exist:nT`.)

`\l__msg_class_tl`
`\l__msg_current_class_tl`

Support variables needed for the redirection system.

```
\tl_new:N \l__msg_class_tl
\tl_new:N \l__msg_current_class_tl
```

(*End definition for* `\l__msg_class_tl` *and* `\l__msg_current_class_tl`. *These variables are documented on page* **??**.)

`\l__msg_redirect_prop`

For redirection of individually-named messages

```
\prop_new:N \l__msg_redirect_prop
```

(*End definition for* `\l__msg_redirect_prop`. *This variable is documented on page* **??**.)

`\l__msg_hierarchy_seq`

During redirection, split the message name into a sequence with items `{/module/submodule}`, `{/module}`, and `{}`.

```
\seq_new:N \l__msg_hierarchy_seq
```

(*End definition for* `\l__msg_hierarchy_seq`. *This variable is documented on page* **??**.)

`\l__msg_class_loop_seq`

Classes encountered when following redirections to check for loops.

```
\seq_new:N \l__msg_class_loop_seq
```

(*End definition for* `\l__msg_class_loop_seq`. *This variable is documented on page* **??**.)

`\__msg_use:nnnnnnn`
`\__msg_use_redirect_name:n`
`\__msg_use_hierarchy:nwwN`
`\__msg_use_redirect_module:n`
`\__msg_use_code:`

Actually using a message is a multi-step process. First, some safety checks on the message and class requested. The code and arguments are then stored to avoid passing them around. The assignment to `\__msg_use_code:` is similar to `\tl_set:Nn`. The message is eventually produced with whatever `\l__msg_class_tl` is when `\__msg_use_code:` is called.

```
\cs_new_protected:Npn \__msg_use:nnnnnnn #1#2#3#4#5#6#7
  {
    \msg_if_exist:nnTF {#2} {#3}
      {
        \__msg_class_chk_exist:nT {#1}
          {
            \tl_set:Nn \l__msg_current_class_tl {#1}
            \cs_set_protected_nopar:Npx \__msg_use_code:
              {
```

```
                \exp_not:n
                  {
                    \use:c { __msg_ \l__msg_class_tl _code:nnnnnn }
                      {#2} {#3} {#4} {#5} {#6} {#7}
                  }
              }
            \__msg_use_redirect_name:n { #2 / #3 }
          }
      }
      { \__msg_kernel_error:nnxx { kernel } { message-unknown } {#2} {#3} }
  }
\cs_new_protected_nopar: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.

```
\cs_new_protected:Npn \__msg_use_redirect_name:n #1
  {
    \prop_get:NnNTF \l__msg_redirect_prop { / #1 } \l__msg_class_tl
      { \__msg_use_code: }
      {
        \seq_clear:N \l__msg_hierarchy_seq
        \__msg_use_hierarchy:nwwN { }
          #1 \q_mark \__msg_use_hierarchy:nwwN
          /  \q_mark \use_none_delimit_by_q_stop:w
          \q_stop
        \__msg_use_redirect_module:n { }
      }
  }
\cs_new_protected:Npn \__msg_use_hierarchy:nwwN #1#2 / #3 \q_mark #4
  {
    \seq_put_left:Nn \l__msg_hierarchy_seq {#1}
    #4 { #1 / #2 } #3 \q_mark #4
  }
```

At this point, the items of `\l__msg_hierarchy_seq` are the various levels at which we should look for a redirection. Redirections which are less specific than the argument of `\__msg_use_redirect_module:n` are not attempted. This argument is empty for a class redirection, `/module` for a module redirection, *etc.* Loop through the sequence to find the most specific redirection, with module `##1`. The loop is interrupted after testing for a redirection for `##1` equal to the argument `#1` (least specific redirection allowed). When a redirection is found, break the mapping, then if the redirection targets the same class, output the code with that class, and otherwise set the target as the new current class, and search for further redirections. Those redirections should be at least as specific as `##1`.

```
\cs_new_protected:Npn \__msg_use_redirect_module:n #1
  {
```

```
    \seq_map_inline:Nn \l__msg_hierarchy_seq
      {
        \prop_get:cnNTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
          {##1} \l__msg_class_tl
          {
            \seq_map_break:n
              {
                \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
                  { \__msg_use_code: }
                  {
                    \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
                    \__msg_use_redirect_module:n {##1}
                  }
              }
          }
          {
            \str_if_eq:nnT {##1} {#1}
              {
                \tl_set_eq:NN \l__msg_class_tl \l__msg_current_class_tl
                \seq_map_break:n { \__msg_use_code: }
              }
          }
      }
  }
```

(*End definition for* `\__msg_use:nnnnnnn`.)

\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.

```
\cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
  {
    \tl_if_empty:nTF {#3}
      { \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 } }
      {
        \__msg_class_chk_exist:nT {#3}
          { \prop_put:Nnn \l__msg_redirect_prop { / #1 / #2 } {#3} }
      }
  }
```

(*End definition for* `\msg_redirect_name:nnn`. *This function is documented on page 146.*)

\msg_redirect_class:nn
\msg_redirect_module:nnn
__msg_redirect:nnn
__msg_redirect_loop_chk:nnn
__msg_redirect_loop_list:n

If the target class is empty, eliminate the corresponding redirection. Otherwise, add the redirection. We must then check for a loop: as an initialization, we start by storing the initial class in `\l__msg_current_class_tl`.

```
\cs_new_protected_nopar:Npn \msg_redirect_class:nn
  { \__msg_redirect:nnn { } }
\cs_new_protected:Npn \msg_redirect_module:nnn #1
  { \__msg_redirect:nnn { / #1 } }
\cs_new_protected:Npn \__msg_redirect:nnn #1#2#3
  {
    \__msg_class_chk_exist:nT {#2}
```

```
      {
        \tl_if_empty:nTF {#3}
          { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
          {
            \__msg_class_chk_exist:nT {#3}
              {
                \prop_put:cnn { l__msg_redirect_ #2 _prop } {#1} {#3}
                \tl_set:Nn \l__msg_current_class_tl {#2}
                \seq_clear:N \l__msg_class_loop_seq
                \__msg_redirect_loop_chk:nnn {#2} {#3} {#1}
              }
          }
      }
  }
```

Since multiple redirections can only happen with increasing specificity, a loop requires that all steps are of the same specificity. The new redirection can thus only create a loop with other redirections for the exact same module, `#1`, and not submodules. After some initialization above, follow redirections with `\l__msg_class_tl`, and keep track in `\l__msg_class_loop_seq` of the various classes encountered. A redirection from a class to itself, or the absence of redirection both mean that there is no loop. A redirection to the initial class marks a loop. To break it, we must decide which redirection to cancel. The user most likely wants the newly added redirection to hold with no further redirection. We thus remove the redirection starting from `#2`, target of the new redirection. Note that no message is emitted by any of the underlying functions: otherwise we may get an infinite loop because of a message from the message system itself.

```
\cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
  {
    \seq_put_right:Nn \l__msg_class_loop_seq {#1}
    \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_class_tl
      {
        \str_if_eq_x:nnF { \l__msg_class_tl } {#1}
          {
            \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl
              {
                \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
                \__msg_kernel_warning:nnxxxx { kernel } { message-redirect-loop }
                  { \seq_item:Nn \l__msg_class_loop_seq { \c_one } }
                  { \seq_item:Nn \l__msg_class_loop_seq { \c_two } }
                  {#3}
                  {
                    \seq_map_function:NN \l__msg_class_loop_seq
                      \__msg_redirect_loop_list:n
                    { \seq_item:Nn \l__msg_class_loop_seq { \c_one } }
                  }
              }
              { \__msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} }
          }
      }
```

```
  }
\cs_generate_variant:Nn \__msg_redirect_loop_chk:nnn { o }
\cs_new:Npn \__msg_redirect_loop_list:n #1 { {#1} ~ => ~ }
```

(*End definition for* `\msg_redirect_class:nn` *and* `\msg_redirect_module:nnn`. *These functions are documented on page 146.*)

17.5 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.

```
\cs_new_protected:Npn \__msg_kernel_new:nnnn #1#2
  { \msg_new:nnnn { LaTeX } { #1 / #2 } }
\cs_new_protected:Npn \__msg_kernel_new:nnn #1#2
  { \msg_new:nnn { LaTeX } { #1 / #2 } }
\cs_new_protected:Npn \__msg_kernel_set:nnnn #1#2
  { \msg_set:nnnn { LaTeX } { #1 / #2 } }
\cs_new_protected:Npn \__msg_kernel_set:nnn #1#2
  { \msg_set:nnn { LaTeX } { #1 / #2 } }
```

(*End definition for* `\__msg_kernel_new:nnnn` *and* `\__msg_kernel_new:nnn`. *These functions are documented on page* ??.)

`\__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.

```
\group_begin:
  \cs_set_protected:Npn \__msg_kernel_class_new:nN #1
    { \__msg_kernel_class_new_aux:nN { kernel_ #1 } }
  \cs_set_protected:Npn \__msg_kernel_class_new_aux:nN #1#2
    {
      \cs_new_protected:cpn { __msg_ #1 :nnnnnn } ##1##2##3##4##5##6
        {
          \use:x
            {
              \exp_not:n { #2 { LaTeX } { ##1 / ##2 } }
                { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
                { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
            }
        }
      \cs_new_protected:cpx { __msg_ #1 :nnnnn } ##1##2##3##4##5
        { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
      \cs_new_protected:cpx { __msg_ #1 :nnnn } ##1##2##3##4
        { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
      \cs_new_protected:cpx { __msg_ #1 :nnn } ##1##2##3
        { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
      \cs_new_protected:cpx { __msg_ #1 :nn } ##1##2
        { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
      \cs_new_protected:cpx { __msg_ #1 :nnxxxx } ##1##2##3##4##5##6
```

```
        {
          \use:x
            {
              \exp_not:N \exp_not:n
                { \exp_not:c { __msg_ #1 :nnnnnn } {##1} {##2} }
                {##3} {##4} {##5} {##6}
            }
        }
      \cs_new_protected:cpx { __msg_ #1 :nnxxx } ##1##2##3##4##5
        { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
      \cs_new_protected:cpx { __msg_ #1 :nnxx } ##1##2##3##4
        { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
      \cs_new_protected:cpx { __msg_ #1 :nnx } ##1##2##3
        { \exp_not:c { __msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
    }
```

(*End definition for* `\__msg_kernel_class_new:nN`.)

__msg_kernel_fatal:nnnnnn
__msg_kernel_fatal:nnnnn
__msg_kernel_fatal:nnnn
__msg_kernel_fatal:nnn
__msg_kernel_fatal:nn
__msg_kernel_fatal:nnxxxx
__msg_kernel_fatal:nnxxx
__msg_kernel_fatal:nnxx
__msg_kernel_fatal:nnx
__msg_kernel_error:nnnnnn
__msg_kernel_error:nnnnn
__msg_kernel_error:nnnn
__msg_kernel_error:nnn
__msg_kernel_error:nn
__msg_kernel_error:nnxxxx
__msg_kernel_error:nnxxx
__msg_kernel_error:nnxx
__msg_kernel_error:nnx

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.

```
    \__msg_kernel_class_new:nN { fatal } \__msg_fatal_code:nnnnnn
    \cs_undefine:N \__msg_kernel_error:nnxx
    \cs_undefine:N \__msg_kernel_error:nnx
    \cs_undefine:N \__msg_kernel_error:nn
    \__msg_kernel_class_new:nN { error } \__msg_error_code:nnnnnn
```

(*End definition for* `\__msg_kernel_fatal:nnnnnn` *and others. These functions are documented on page* ??.)

__msg_kernel_warning:nnnnnn
__msg_kernel_warning:nnnnn
__msg_kernel_warning:nnnn
__msg_kernel_warning:nnn
__msg_kernel_warning:nn
__msg_kernel_warning:nnxxxx
__msg_kernel_warning:nnxxx
__msg_kernel_warning:nnxx
__msg_kernel_warning:nnx
__msg_kernel_info:nnnnnn
__msg_kernel_info:nnnnn
__msg_kernel_info:nnnn
__msg_kernel_info:nnn
__msg_kernel_info:nn
__msg_kernel_info:nnxxxx
__msg_kernel_info:nnxxx
__msg_kernel_info:nnxx
__msg_kernel_info:nnx

Kernel messages which can be redirected simply use the machinery for normal messages, with the module name "LaTeX".

```
    \__msg_kernel_class_new:nN { warning } \msg_warning:nnxxxx
    \__msg_kernel_class_new:nN { info } \msg_info:nnxxxx
```

(*End definition for* `\__msg_kernel_warning:nnnnnn` *and others. These functions are documented on page* ??.)

End the group to eliminate `\__msg_kernel_class_new:nN`.

```
\group_end:
```

Error messages needed to actually implement the message system itself.

```
\__msg_kernel_new:nnnn { kernel } { message-already-defined }
  { Message~'#2'~for~module~'#1'~already~defined. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~was~asked~to~define~a~new~message~called~'#2'\
    by~the~module~'#1':~this~message~already~exists.
    \c_msg_return_text_tl
  }
\__msg_kernel_new:nnnn { kernel } { message-unknown }
  { Unknown~message~'#2'~for~module~'#1'. }
  {
```

```
    \c_msg_coding_error_text_tl
    LaTeX~was~asked~to~display~a~message~called~'#2'\\
    by~the~module~'#1':~this~message~does~not~exist.
    \c_msg_return_text_tl
  }
\__msg_kernel_new:nnnn { kernel } { message-class-unknown }
  { Unknown~message~class~'#1'. }
  {
    LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\\
    this~was~never~defined.
    \c_msg_return_text_tl
  }
\__msg_kernel_new:nnnn { kernel } { message-redirect-loop }
  {
    Message~redirection~loop~caused~by~ {#1} ~=>~ {#2}
    \tl_if_empty:nF {#3} { ~for~module~' \use_none:n #3 ' } .
  }
  {
    Adding~the~message~redirection~ {#1} ~=>~ {#2}
    \tl_if_empty:nF {#3} { ~for~the~module~' \use_none:n #3 ' } ~
    created~an~infinite~loop\\\\
    \iow_indent:n { #4 \\\\ }
  }
```

Messages for earlier kernel modules.

```
\__msg_kernel_new:nnnn { kernel } { bad-number-of-arguments }
  { Function~'#1'~cannot~be~defined~with~#2~arguments. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~define~a~function~'#1'~with~
    #2~arguments.~
    TeX~allows~between~0~and~9~arguments~for~a~single~function.
  }
\__msg_kernel_new:nnnn { kernel } { command-already-defined }
  { Control~sequence~#1~already~defined. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~create~a~new~control~sequence~'#1'~
    but~this~name~has~already~been~used~elsewhere. \\ \\
    The~current~meaning~is:\\
    \ \ #2
  }
\__msg_kernel_new:nnnn { kernel } { command-not-defined }
  { Control~sequence~#1~undefined. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~use~a~command~#1,~but~this~has~not~
    been~defined~yet.
  }
\__msg_kernel_new:nnnn { kernel } { empty-search-pattern }
```

```
  { Empty~search~pattern. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'#1':~that~
    would~lead~to~an~infinite~loop!
  }
\__msg_kernel_new:nnnn { kernel } { out-of-registers }
  { No~room~for~a~new~#1. }
  {
    TeX~only~supports~\int_use:N \c_max_register_int \
    of~each~type.~All~the~#1~registers~have~been~used.~
    This~run~will~be~aborted~now.
  }
\__msg_kernel_new:nnnn { kernel } { missing-colon }
  { Function~'#1'~contains~no~':'. }
  {
    \c_msg_coding_error_text_tl
    Code-level~functions~must~contain~':'~to~separate~the~
    argument~specification~from~the~function~name.~This~is~
    needed~when~defining~conditionals~or~variants,~or~when~building~a~
    parameter~text~from~the~number~of~arguments~of~the~function.
  }
\__msg_kernel_new:nnnn { kernel } { protected-predicate }
  { Predicate~'#1'~must~be~expandable. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~define~'#1'~as~a~protected~predicate.~
    Only~expandable~tests~can~have~a~predicate~version.
  }
\__msg_kernel_new:nnnn { kernel } { conditional-form-unknown }
  { Conditional~form~'#1'~for~function~'#2'~unknown. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~define~the~conditional~form~'#1'~of~
    the~function~'#2',~but~only~'TF',~'T',~'F',~and~'p'~forms~exist.
  }
\bool_if:NT \l@expl@check@declarations@bool
  {
    \__msg_kernel_new:nnnn { check } { non-declared-variable }
      { The~variable~#1~has~not~been~declared~\msg_line_context:. }
      {
        Checking~is~active,~and~you~have~tried~do~so~something~like: \\
        \ \ \tl_set:Nn ~ #1 ~ \{ ~ ... ~ \} \\
        without~first~having: \\
        \ \ \tl_new:N ~ #1  \\
        \\
        LaTeX~will~create~the~variable~and~continue.
      }
  }
\__msg_kernel_new:nnnn { kernel } { scanmark-already-defined }
```

```
  { Scan~mark~#1~already~defined. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~create~a~new~scan~mark~'#1'~
    but~this~name~has~already~been~used~for~a~scan~mark.
  }
\__msg_kernel_new:nnnn { kernel } { variable-not-defined }
  { Variable~#1~undefined. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~show~a~variable~#1,~but~this~has~not~
    been~defined~yet.
  }
\__msg_kernel_new:nnnn { kernel } { variant-too-long }
  { Variant~form~'#1'~longer~than~base~signature~of~'#2'. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
    with~a~signature~starting~with~'#1',~but~that~is~longer~than~
    the~signature~(part~after~the~colon)~of~'#2'.
  }
\__msg_kernel_new:nnnn { kernel } { invalid-variant }
  { Variant~form~'#1'~invalid~for~base~form~'#2'. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
    with~a~signature~starting~with~'#1',~but~cannot~change~an~argument~
    from~type~'#3'~to~type~'#4'.
  }
```

Some errors only appear in expandable settings, hence don't need a "more-text" argument.

```
\__msg_kernel_new:nnn { kernel } { bad-variable }
  { Erroneous~variable~#1 used! }
\__msg_kernel_new:nnn { kernel } { misused-sequence }
  { A~sequence~was~misused. }
\__msg_kernel_new:nnn { kernel } { misused-prop }
  { A~property~list~was~misused. }
\__msg_kernel_new:nnn { kernel } { negative-replication }
  { Negative~argument~for~\prg_replicate:nn. }
\__msg_kernel_new:nnn { kernel } { unknown-comparison }
  { Relation~symbol~'#1'~unknown:~use~=,~<,~>,~==,~!=,~<=,~>=. }
\__msg_kernel_new:nnn { kernel } { zero-step }
  { Zero~step~size~for~step~function~#1. }
```

Messages used by the "`show`" functions.

```
\__msg_kernel_new:nnn { kernel } { show-clist }
  {
    The~comma~list~
    \str_if_eq:nnF {#1} { \l__clist_internal_clist } { \token_to_str:N #1~}
```

```
    \clist_if_empty:NTF #1
      { is~empty }
      { contains~the~items~(without~outer~braces): }
  }
\__msg_kernel_new:nnn { kernel } { show-prop }
  {
    The~property~list~\token_to_str:N #1~
    \prop_if_empty:NTF #1
      { is~empty }
      { contains~the~pairs~(without~outer~braces): }
  }
\__msg_kernel_new:nnn { kernel } { show-seq }
  {
    The~sequence~\token_to_str:N #1~
    \seq_if_empty:NTF #1
      { is~empty }
      { contains~the~items~(without~outer~braces): }
  }
\__msg_kernel_new:nnn { kernel } { show-no-stream }
  { No~ #1 ~streams~are~open }
\__msg_kernel_new:nnn { kernel } { show-open-streams }
  { The~following~ #1 ~streams~are~in~use: }
```

17.6 Expandable errors

__msg_expandable_error:n
__msg_expandable_error:w

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 `\c_zero` prevents losing braces around the user-inserted text if any, and stops the expansion of `\romannumeral`.

```
\group_begin:
\char_set_catcode_math_superscript:N \^
\char_set_lccode:nn { `^ } { `\  }
\char_set_lccode:nn { `L } { `L }
\char_set_lccode:nn { `T } { `T }
\char_set_lccode:nn { `X } { `X }
\tl_to_lowercase:n
  {
    \cs_new:Npx \__msg_expandable_error:n #1
      {
        \exp_not:n
```

```
          {
            \tex_romannumeral:D
            \exp_after:wN \exp_after:wN
            \exp_after:wN \__msg_expandable_error:w
            \exp_after:wN \exp_after:wN
            \exp_after:wN \c_zero
          }
        \exp_not:N \use:n { \exp_not:c { LaTeX3~error: } ^ #1 } ^
      }
    \cs_new:Npn \__msg_expandable_error:w #1 ^ #2 ^ { #1 }
  }
\group_end:
```

(*End definition for* `\__msg_expandable_error:n`.)

__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`.

```
\cs_new:Npn \__msg_kernel_expandable_error:nnnnnn #1#2#3#4#5#6
  {
    \exp_args:Nf \__msg_expandable_error:n
      {
        \exp_args:NNc \exp_after:wN \exp_stop_f:
          { \c__msg_text_prefix_tl LaTeX / #1 / #2 }
          {#3} {#4} {#5} {#6}
      }
  }
\cs_new:Npn \__msg_kernel_expandable_error:nnnnn #1#2#3#4#5
  {
    \__msg_kernel_expandable_error:nnnnnn
      {#1} {#2} {#3} {#4} {#5} { }
  }
\cs_new:Npn \__msg_kernel_expandable_error:nnnn #1#2#3#4
  {
    \__msg_kernel_expandable_error:nnnnnn
      {#1} {#2} {#3} {#4} { } { }
  }
\cs_new:Npn \__msg_kernel_expandable_error:nnn #1#2#3
  {
    \__msg_kernel_expandable_error:nnnnnn
      {#1} {#2} {#3} { } { } { }
  }
\cs_new:Npn \__msg_kernel_expandable_error:nn #1#2
  {
    \__msg_kernel_expandable_error:nnnnnn
      {#1} {#2} { } { } { } { }
  }
```

(*End definition for* `\__msg_kernel_expandable_error:nnnnnn` *and others. These functions are documented on page* **??**.)

17.7 Showing variables

Functions defined in this section are used for diagnostic functions in l3clist, l3file, l3prop, l3seq, xtemplate

__msg_term:nnnnnn
__msg_term:nnnnnV
__msg_term:nnnnn
__msg_term:nnn
__msg_term:nn

Print the text of a message to the terminal without formatting: short cuts around \iow_wrap:nnnN.

```
\cs_new_protected:Npn \__msg_term:nnnnnn #1#2#3#4#5#6
  {
    \iow_wrap:nnnN
      { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
      { } { } \iow_term:n
  }
\cs_generate_variant:Nn \__msg_term:nnnnnn { nnnnnV }
\cs_new_protected:Npn \__msg_term:nnnnn #1#2#3#4#5
  { \__msg_term:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }
\cs_new_protected:Npn \__msg_term:nnn #1#2#3
  { \__msg_term:nnnnnn {#1} {#2} {#3} { } { } { } }
\cs_new_protected:Npn \__msg_term:nn #1#2
  { \__msg_term:nnnnnn {#1} {#2} { } { } { } { } }
```

(*End definition for* __msg_term:nnnnnn *and* __msg_term:nnnnnV*.*)

__msg_show_variable:Nnn
__msg_show_variable:n
__msg_show_variable_aux:n
__msg_show_variable_aux:w

The arguments of __msg_show_variable:Nnn are

- The ⟨*variable*⟩ to be shown.
- The type of the variable.
- A mapping of the form \seq_map_function:NN ⟨*variable*⟩ __msg_show_item:n, which produces the formatted string.

As for __kernel_register_show:N, check that the variable is defined. If it is, output the introductory message, then show the contents #3 using __msg_show_variable:n. This wraps the contents (with leading >␣) to a fixed number of characters per line. The expansion of #3 may either be empty or start with >␣. A leading >, if present, is removed using a w-type auxiliary, as well as a space following it (via f-expansion). Note that we cannot remove the space as a delimiter for the w-type auxiliary, because a line-break may be taken there, and the space would then disappear. Finally, the resulting token list \l__msg_internal_tl is displayed to the terminal, with an odd \exp_after:wN which expands the closing brace to improve the output slightly.

```
\cs_new_protected:Npn \__msg_show_variable:Nnn #1#2#3
  {
    \cs_if_exist:NTF #1
      {
        \__msg_term:nnn { LaTeX / kernel } { show- #2 } {#1}
        \__msg_show_variable:n {#3}
      }
      {
        \__msg_kernel_error:nnx { kernel } { variable-not-defined }
```

```
          { \token_to_str:N #1 }
      }
  }
\cs_new_protected:Npn \__msg_show_variable:n #1
  { \iow_wrap:nnnN {#1} { } { } \__msg_show_variable_aux:n }
\cs_new_protected:Npn \__msg_show_variable_aux:n #1
  {
    \tl_if_empty:nTF {#1}
      { \tl_clear:N \l__msg_internal_tl }
      { \tl_set:Nf \l__msg_internal_tl { \__msg_show_variable_aux:w #1 } }
    \etex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
      { \exp_after:wN \l__msg_internal_tl }
  }
\cs_new:Npn \__msg_show_variable_aux:w #1 > { }
```

(*End definition for* `\__msg_show_variable:Nnn`*.*)

`\__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.

```
\cs_new:Npn \__msg_show_item:n #1
  {
    \\ > \ \ \{ \tl_to_str:n {#1} \}
  }
\cs_new:Npn \__msg_show_item:nn #1#2
  {
    \\ > \ \ \{ \tl_to_str:n {#1} \}
    \ \ => \ \ \{ \tl_to_str:n {#2} \}
  }
\cs_new:Npn \__msg_show_item_unbraced:nn #1#2
  {
    \\ > \ \ \tl_to_str:n {#1}
    \ \ => \ \ \tl_to_str:n {#2}
  }
```

(*End definition for* `\__msg_show_item:n`*.*)

```
⟨/initex | package⟩
```

18 l3keys Implementation

```
⟨*initex | package⟩
```

18.1 Low-level interface

```
⟨@@=keyval⟩
```

For historical reasons this code uses the 'keyval' module prefix.

`\g__keyval_level_int`

To allow nesting of `\keyval_parse:NNn`, an integer is needed for the current level.

```
\int_new:N \g__keyval_level_int
```

(*End definition for* `\g__keyval_level_int`*. This variable is documented on page* **??***.*)

\l__keyval_key_tl
\l__keyval_value_tl

The current key name and value.

```
\tl_new:N \l__keyval_key_tl
\tl_new:N \l__keyval_value_tl
```

(*End definition for* `\l__keyval_key_tl` *and* `\l__keyval_value_tl`*. These variables are documented on page* **??**.)

\l__keyval_sanitise_tl
\l__keyval_parse_tl

Token list variables for dealing with awkward category codes in the input.

```
\tl_new:N \l__keyval_sanitise_tl
\tl_new:N \l__keyval_parse_tl
```

(*End definition for* `\l__keyval_sanitise_tl`*. This variable is documented on page* **??**.)

__keyval_parse:n

The parsing function first deals with the category codes for `=` and `,`, so that there are no odd events. The input is then handed off to the element by element system.

```
\group_begin:
  \char_set_catcode_active:n { `\= }
  \char_set_catcode_active:n { `\, }
  \char_set_lccode:nn { `\8 } { `\= }
  \char_set_lccode:nn { `\9 } { `\, }
\tl_to_lowercase:n
  {
    \group_end:
    \cs_new_protected:Npn \__keyval_parse:n #1
      {
        \group_begin:
          \tl_set:Nn \l__keyval_sanitise_tl {#1}
          \tl_replace_all:Nnn \l__keyval_sanitise_tl { = } { 8 }
          \tl_replace_all:Nnn \l__keyval_sanitise_tl { , } { 9 }
          \tl_clear:N \l__keyval_parse_tl
          \exp_after:wN \__keyval_parse_elt:w \exp_after:wN
            \q_nil \l__keyval_sanitise_tl 9 \q_recursion_tail 9 \q_recursion_stop
        \exp_after:wN \group_end:
        \l__keyval_parse_tl
      }
  }
```

(*End definition for* `\__keyval_parse:n`*. This function is documented on page* **??**.)

__keyval_parse_elt:w

Each item to be parsed will have `\q_nil` added to the front. Hence the blank test here can always be used to find a totally empty argument. To allow rapid matching for an `=` while not stripping any braces, another `\q_nil` needed before the next phase of the parser. Finally, loop around for the next item, adding in the `\q_nil`: this happens whatever the nature of the current argument as the end-of-recursion will clear up in all cases.

```
\cs_new_protected:Npn \__keyval_parse_elt:w #1 ,
  {
    \tl_if_blank:oF { \use_none:n #1 }
      {
        \quark_if_recursion_tail_stop:o { \use_none:n #1 }
        \__keyval_split_key_value:w #1 \q_nil = = \q_stop
      }
```

```
    \__keyval_parse_elt:w \q_nil
  }
```

(*End definition for* `\__keyval_parse_elt:w`. *This function is documented on page* **??**.)

`\__keyval_split_key_value:w`
`\__keyval_split_key:w`

Split the key and value using a delimited argument. The `\q_nil` values added earlier ensure that no braces will be stripped as part of this process. A blank test can then be used on `#3`: it is only empty if there was no `=` in the original input. In that case, strip a `\q_nil` from the end of the key name then hand on to remove other things and store as `\l__keyval_key_tl` before adding to the output token list. In the case where there is an `=`, first tidy up the key, this time without a trailing `\q_nil`, then do a check to ensure that `#3` is exactly one token (`=`). With that done, the final stage is to hand off to tidy up the value.

```
\cs_new_protected:Npn \__keyval_split_key_value:w #1 = #2 = #3 \q_stop
  {
    \tl_if_blank:nTF {#3}
      {
        \__keyval_split_key:w #1 \q_stop
        \tl_put_right:Nx \l__keyval_parse_tl
          {
            \exp_not:c
              { __keyval_key_no_value_elt_ \int_use:N \g__keyval_level_int :n }
              { \exp_not:o \l__keyval_key_tl }
          }
      }
      {
        \__keyval_split:Nn \l__keyval_key_tl {#1}
        \tl_if_blank:oTF { \use_none:n #3 }
          { \__keyval_split_value:w \q_nil #2 \q_stop }
          { \__msg_kernel_error:nn { kernel } { misplaced-equals-sign } }
      }
  }
\cs_new_protected:Npn \__keyval_split_key:w #1 \q_nil \q_stop
  { \__keyval_split:Nn \l__keyval_key_tl {#1} }
```

(*End definition for* `\__keyval_split_key_value:w`. *This function is documented on page* **??**.)

`\__keyval_split:Nn`
`\__keyval_split:Nw`

There are two possible cases here. The first case is that `#1` is surrounded by braces, in which case the `\use_none:nnn #1 \q_nil \q_nil` will yield `\q_nil`. There, we can remove the leading `\q_nil`, the braces and any spaces around the outside with `\use_-ii:nnn`. On the other hand, if there are no braces then the second branch removes the leading `\q_nil` and any surrounding spaces.

```
\cs_new_protected:Npn \__keyval_split:Nn #1#2
  {
    \quark_if_nil:oTF { \use_none:nnn #2 \q_nil \q_nil }
      { \tl_set:Nx #1 { \exp_not:o { \use_ii:nnn #2 \q_nil } } }
      { \__keyval_split:Nw #1 #2 \q_stop }
  }
\cs_new_protected:Npn \__keyval_split:Nw #1 \q_nil #2 \q_stop
  { \tl_set:Nx #1 { \tl_trim_spaces:n {#2} } }
```

(*End definition for* `\__keyval_split:Nn`. *This function is documented on page* **??**.)

`\__keyval_split_value:w`

As this stage there is just the value to deal with. The leading and trailing `\q_nil` tokens are removed in two steps before storing the value with spaces stripped (see `\__keyval_split:Nn`). Doing the storage of key and value in one shot will put exactly the right number of brace groups into the output.

```
\cs_new_protected:Npn \__keyval_split_value:w #1 \q_nil \q_stop
  {
    \__keyval_split:Nn \l__keyval_value_tl {#1}
    \tl_put_right:Nx \l__keyval_parse_tl
      {
        \exp_not:c
          { __keyval_key_value_elt_ \int_use:N \g__keyval_level_int :nn }
          { \exp_not:o \l__keyval_key_tl }
          { \exp_not:o \l__keyval_value_tl }
      }
  }
```

(*End definition for* `\__keyval_split_value:w`. *This function is documented on page* **??**.)

`\keyval_parse:NNn`

The outer parsing routine just sets up the processing functions and hands off.

```
\cs_new_protected:Npn \keyval_parse:NNn #1#2#3
  {
    \int_gincr:N \g__keyval_level_int
    \cs_gset_eq:cN
      { __keyval_key_no_value_elt_ \int_use:N \g__keyval_level_int :n } #1
    \cs_gset_eq:cN
      { __keyval_key_value_elt_ \int_use:N \g__keyval_level_int :nn }   #2
    \__keyval_parse:n {#3}
    \int_gdecr:N \g__keyval_level_int
  }
```

(*End definition for* `\keyval_parse:NNn`. *This function is documented on page 161.*)

One message for the low level parsing system.

```
\__msg_kernel_new:nnnn { kernel } { misplaced-equals-sign }
  { Misplaced~equals~sign~in~key-value~input~\msg_line_number: }
  {
    LaTeX~is~attempting~to~parse~some~key-value~input~but~found~
    two~equals~signs~not~separated~by~a~comma.
  }
```

18.2 Constants and variables

```
⟨@@=keys⟩
```

`\c__keys_code_root_tl`
`\c__keys_info_root_tl`

The prefixes for the code and variables of the keys themselves.

```
\tl_const:Nn \c__keys_code_root_tl { key~code~>~ }
\tl_const:Nn \c__keys_info_root_tl { key~info~>~ }
```

(*End definition for* `\c__keys_code_root_tl` *and* `\c__keys_info_root_tl`. *These variables are documented on page* **??**.)

\c__keys_props_root_tl The prefix for storing properties.

```
\tl_const:Nn \c__keys_props_root_tl { key~prop~>~ }
```

(*End definition for* `\c__keys_props_root_tl`. *This variable is documented on page* **??**.)

\l_keys_choice_int
\l_keys_choice_tl Publicly accessible data on which choice is being used when several are generated as a set.

```
\int_new:N \l_keys_choice_int
\tl_new:N \l_keys_choice_tl
```

(*End definition for* `\l_keys_choice_int` *and* `\l_keys_choice_tl`. *These variables are documented on page 156*.)

\l__keys_groups_clist Used for storing and recovering the list of groups which apply to a key: set as a comma list but at one point we have to use this for a token list recovery.

```
\clist_new:N \l__keys_groups_clist
```

(*End definition for* `\l__keys_groups_clist`. *This variable is documented on page* **??**.)

\l_keys_key_tl The name of a key itself: needed when setting keys.

```
\tl_new:N \l_keys_key_tl
```

(*End definition for* `\l_keys_key_tl`. *This variable is documented on page 158*.)

\l__keys_module_tl The module for an entire set of keys.

```
\tl_new:N \l__keys_module_tl
```

(*End definition for* `\l__keys_module_tl`. *This variable is documented on page* **??**.)

\l__keys_no_value_bool A marker is needed internally to show if only a key or a key plus a value was seen: this is recorded here.

```
\bool_new:N \l__keys_no_value_bool
```

(*End definition for* `\l__keys_no_value_bool`. *This variable is documented on page* **??**.)

\l__keys_only_known_bool Used to track if only “known” keys are being set.

```
\bool_new:N \l__keys_only_known_bool
```

(*End definition for* `\l__keys_only_known_bool`. *This variable is documented on page* **??**.)

\l_keys_path_tl The “path” of the current key is stored here: this is available to the programmer and so is public.

```
\tl_new:N \l_keys_path_tl
```

(*End definition for* `\l_keys_path_tl`. *This variable is documented on page 158*.)

\l__keys_property_tl The “property” begin set for a key at definition time is stored here.

```
\tl_new:N \l__keys_property_tl
```

(*End definition for* `\l__keys_property_tl`. *This variable is documented on page* **??**.)

\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”).

```
\bool_new:N \l__keys_selective_bool
\bool_new:N \l__keys_filtered_bool
```

(*End definition for* `\l__keys_selective_bool` *and* `\l__keys_filtered_bool`*. These variables are documented on page* **??**.)

`\l__keys_selective_seq` The list of key groups being filtered in or out during selective setting.

```
\seq_new:N \l__keys_selective_seq
```

(*End definition for* `\l__keys_selective_seq`*. This variable is documented on page* **??**.)

`\l__keys_unused_clist` Used when setting only some keys to store those left over.

```
\tl_new:N \l__keys_unused_clist
```

(*End definition for* `\l__keys_unused_clist`*. This variable is documented on page* **??**.)

`\l_keys_value_tl` The value given for a key: may be empty if no value was given.

```
\tl_new:N \l_keys_value_tl
```

(*End definition for* `\l_keys_value_tl`*. This variable is documented on page 158.*)

`\l__keys_tmp_bool` Scratch space.

```
\bool_new:N \l__keys_tmp_bool
```

(*End definition for* `\l__keys_tmp_bool`*. This variable is documented on page* **??**.)

18.3 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).

```
\cs_new_protected:Npn \keys_define:nn
  { \__keys_define:onn \l__keys_module_tl }
\cs_new_protected:Npn \__keys_define:nnn #1#2#3
  {
    \tl_set:Nx \l__keys_module_tl { \tl_to_str:n {#2} }
    \keyval_parse:NNn \__keys_define_elt:n \__keys_define_elt:nn {#3}
    \tl_set:Nn \l__keys_module_tl {#1}
  }
\cs_generate_variant:Nn \__keys_define:nnn { o }
```

(*End definition for* `\keys_define:nn`*. This function is documented on page 151.*)

`\__keys_define_elt:n`
`\__keys_define_elt:nn`
`\__keys_define_elt_aux:nn`

The outer functions here record whether a value was given and then converge on a common internal mechanism. There is first a search for a property in the current key name, then a check to make sure it is known before the code hands off to the next step.

```
\cs_new_protected:Npn \__keys_define_elt:n #1
  {
    \bool_set_true:N \l__keys_no_value_bool
    \__keys_define_elt_aux:nn {#1} { }
  }
\cs_new_protected:Npn \__keys_define_elt:nn #1#2
  {
    \bool_set_false:N \l__keys_no_value_bool
    \__keys_define_elt_aux:nn {#1} {#2}
  }
```

```
\cs_new_protected:Npn \__keys_define_elt_aux:nn #1#2
  {
    \__keys_property_find:n {#1}
    \cs_if_exist:cTF { \c__keys_props_root_tl \l__keys_property_tl }
      { \__keys_define_key:n {#2} }
      {
        \str_if_eq_x:nnF { \l__keys_property_tl } { .abort: }
          {
            \__msg_kernel_error:nnxx { kernel } { property-unknown }
              { \l__keys_property_tl } { \l_keys_path_tl }
          }
      }
  }
```

(*End definition for* `\__keys_define_elt:n`.)

`\__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.

```
\cs_new_protected:Npn \__keys_property_find:n #1
  {
    \tl_set:Nx \l_keys_path_tl { \l__keys_module_tl / }
    \tl_if_in:nnTF {#1} { . }
      { \__keys_property_find:w #1 \q_stop }
      {
        \__msg_kernel_error:nnx { kernel } { key-no-property } {#1}
        \tl_set:Nn \l__keys_property_tl { .abort: }
      }
  }
\cs_new_protected:Npn \__keys_property_find:w #1 . #2 \q_stop
  {
    \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl \tl_to_str:n {#1} }
    \tl_if_in:nnTF {#2} { . }
      {
        \tl_set:Nx \l_keys_path_tl { \l_keys_path_tl . }
        \__keys_property_find:w #2 \q_stop
      }
      { \tl_set:Nn \l__keys_property_tl { . #2 } }
  }
```

(*End definition for* `\__keys_property_find:n`.)

`\__keys_define_key:n`
`\__keys_define_key: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.

```
\cs_new_protected:Npn \__keys_define_key:n #1
  {
    \bool_if:NTF \l__keys_no_value_bool
      {
        \exp_after:wN \__keys_define_key:w
```

```
        \l__keys_property_tl \q_stop
        { \use:c { \c__keys_props_root_tl \l__keys_property_tl } }
        {
          \__msg_kernel_error:nnxx { kernel }
            { property-requires-value } { \l__keys_property_tl }
            { \l_keys_path_tl }
          }
      }
      { \use:c { \c__keys_props_root_tl \l__keys_property_tl } {#1} }
  }
\cs_new_protected:Npn \__keys_define_key:w #1 : #2 \q_stop
  { \tl_if_empty:nTF {#2} }
```

(*End definition for* `\__keys_define_key:n`*.*)

18.4 Turning properties into actions

`\__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.

```
\cs_new_protected:Npn \__keys_bool_set:Nn #1#2
  {
    \bool_if_exist:NF #1 { \bool_new:N #1 }
    \__keys_choice_make:
    \__keys_cmd_set:nx { \l_keys_path_tl / true }
      { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
    \__keys_cmd_set:nx { \l_keys_path_tl / false }
      { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
    \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
      {
        \__msg_kernel_error:nnx { kernel } { boolean-values-only }
          { \l_keys_key_tl }
      }
    \__keys_default_set:n { true }
  }
\cs_generate_variant:Nn \__keys_bool_set:Nn { c }
```

(*End definition for* `\__keys_bool_set:Nn` *and* `\__keys_bool_set:cn`*.*)

`\__keys_bool_set_inverse:Nn`
`\__keys_bool_set_inverse:cn`

Inverse boolean setting is much the same.

```
\cs_new_protected:Npn \__keys_bool_set_inverse:Nn #1#2
  {
    \bool_if_exist:NF #1 { \bool_new:N #1 }
    \__keys_choice_make:
    \__keys_cmd_set:nx { \l_keys_path_tl / true }
      { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
    \__keys_cmd_set:nx { \l_keys_path_tl / false }
      { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
    \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
      {
        \__msg_kernel_error:nnx { kernel } { boolean-values-only }
          { \l_keys_key_tl }
```

```
      }
    \__keys_default_set:n { true }
  }
\cs_generate_variant:Nn \__keys_bool_set_inverse:Nn { c }
```

(*End definition for* `\__keys_bool_set_inverse:Nn` *and* `\__keys_bool_set_inverse:cn`*.*)

`\__keys_choice_make:` To make a choice from a key, two steps: set the code, and set the unknown key.

```
\cs_new_protected_nopar:Npn \__keys_choice_make:
  {
    \__keys_cmd_set:nn { \l_keys_path_tl }
      { \__keys_choice_find:n {##1} }
    \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
      {
        \__msg_kernel_error:nnxx { kernel } { key-choice-unknown }
          { \l_keys_path_tl } {##1}
      }
  }
```

(*End definition for* `\__keys_choice_make:`*.*)

`\__keys_choices_make:nn` Auto-generating choices means setting up the root key as a choice, then defining each choice in turn.

```
\cs_new_protected:Npn \__keys_choices_make:nn #1#2
  {
    \__keys_choice_make:
    \int_zero:N \l_keys_choice_int
    \clist_map_inline:nn {#1}
      {
        \int_incr:N \l_keys_choice_int
        \__keys_cmd_set:nx { \l_keys_path_tl / ##1 }
          {
            \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
            \int_set:Nn \exp_not:N \l_keys_choice_int
              { \int_use:N \l_keys_choice_int }
            \exp_not:n {#2}
          }
      }
  }
```

(*End definition for* `\__keys_choices_make:nn`*.*)

`\__keys_cmd_set:nn`
`\__keys_cmd_set:nx`
`\__keys_cmd_set:Vn`
`\__keys_cmd_set:Vo`

Creating a new command means tidying up the properties and then making the internal function which actually does the work.

```
\cs_new_protected:Npn \__keys_cmd_set:nn #1#2
  {
    \prop_clear_new:c { \c__keys_info_root_tl #1 }
    \cs_set:cpn { \c__keys_code_root_tl #1 } ##1 {#2}
  }
\cs_generate_variant:Nn \__keys_cmd_set:nn { nx , Vn , Vo }
```

(*End definition for* `\__keys_cmd_set:nn` *and others.*)

`\__keys_default_set:n` Setting a default value is easy.

```
\cs_new_protected:Npn \__keys_default_set:n #1
  {
    \prop_if_exist:cT { \c__keys_info_root_tl \l_keys_path_tl }
      { \prop_put:cnn { \c__keys_info_root_tl \l_keys_path_tl } { default } {#1} }
  }
```

(*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.

```
\cs_new_protected:Npn \__keys_groups_set:n #1
  {
    \prop_if_exist:cT { \c__keys_info_root_tl \l_keys_path_tl }
      {
        \clist_set:Nn \l__keys_groups_clist {#1}
        \prop_put:cnV { \c__keys_info_root_tl \l_keys_path_tl }
          { groups } \l__keys_groups_clist
      }
  }
```

(*End definition for* `\__keys_groups_set:n`.)

`\__keys_initialise:n`
`\__keys_initialise:wn`
A set up for initialisation from which the key system requires that the path is split up into a module and a key name. At this stage, `\l_keys_path_tl` will contain `/` so a split is easy to do.

```
\cs_new_protected:Npn \__keys_initialise:n #1
  { \exp_after:wN \__keys_initialise:wn \l_keys_path_tl \q_stop {#1} }
\cs_new_protected:Npn \__keys_initialise:wn #1 / #2 \q_stop #3
  { \keys_set:nn {#1} { #2 = {#3} } }
```

(*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.

```
\cs_new_protected:Npn \__keys_meta_make:n #1
  {
    \__keys_cmd_set:Vo \l_keys_path_tl
      { \exp_after:wN \keys_set:nn \exp_after:wN { \l__keys_module_tl } {#1} }
  }
\cs_new_protected:Npn \__keys_meta_make:nn #1#2
  { \__keys_cmd_set:Vn \l_keys_path_tl { \keys_set:nn {#1} {#2} } }
```

(*End definition for* `\__keys_meta_make:n`.)

`\__keys_multichoice_find:n`
`\__keys_multichoice_make:`
`\__keys_multichoices_make:nn`
Choices where several values can be selected are very similar to normal exclusive choices. There is just a slight change in implementation to map across a comma-separated list. This then requires that the appropriate set up takes place elsewhere.

```
\cs_new:Npn \__keys_multichoice_find:n #1
  { \clist_map_function:nN {#1} \__keys_choice_find:n }
\cs_new_protected_nopar:Npn \__keys_multichoice_make:
  {
    \__keys_cmd_set:nn { \l_keys_path_tl }
```

```
      { \__keys_multichoice_find:n {##1} }
    \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
      {
        \__msg_kernel_error:nnxx { kernel } { key-choice-unknown }
          { \l_keys_path_tl } {##1}
      }
  }
\cs_new_protected:Npn \__keys_multichoices_make:nn #1#2
  {
    \__keys_multichoice_make:
    \int_zero:N \l_keys_choice_int
    \clist_map_inline:nn {#1}
      {
        \int_incr:N \l_keys_choice_int
        \__keys_cmd_set:nx { \l_keys_path_tl / ##1 }
          {
            \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
            \int_set:Nn \exp_not:N \l_keys_choice_int
              { \int_use:N \l_keys_choice_int }
            \exp_not:n {#2}
          }
      }
  }
```

(*End definition for* `\__keys_multichoice_find:n`.)

`\__keys_value_requirement:n`

Values can be required or forbidden by having the appropriate marker set. First, both the required and forbidden ones are clear, just in case!

```
\cs_new_protected:Npn \__keys_value_requirement:n #1
  {
    \prop_if_exist:cT { \c__keys_info_root_tl \l_keys_path_tl }
      {
        \prop_remove:cn { \c__keys_info_root_tl \l_keys_path_tl } { required }
        \prop_remove:cn { \c__keys_info_root_tl \l_keys_path_tl } { forbidden }
        \prop_put:cnn { \c__keys_info_root_tl \l_keys_path_tl } {#1} { true }
    }
  }
```

(*End definition for* `\__keys_value_requirement:n`.)

`\__keys_variable_set:NnnN`
`\__keys_variable_set:cnnN`

Setting a variable takes the type and scope separately so that it is easy to make a new variable if needed.

```
\cs_new_protected:Npn \__keys_variable_set:NnnN #1#2#3#4
  {
    \use:c { #2_if_exist:NF } #1 { \use:c { #2 _new:N } #1 }
    \__keys_cmd_set:nx { \l_keys_path_tl }
      { \exp_not:c { #2 _ #3 set:N #4 } \exp_not:N #1 \exp_not:n  { {##1} } }
  }
\cs_generate_variant:Nn \__keys_variable_set:NnnN { c }
```

(*End definition for* `\__keys_variable_set:NnnN` *and* `\__keys_variable_set:cnnN`.)

18.5 Creating key properties

The key property functions are all wrappers for internal functions, meaning that things stay readable and can also be altered later on.

Importantly, while key properties have "normal" argument specs, the underlying code always supplies one braced argument to these. As such, argument expansion is handled by hand rather than using the standard tools. This shows up particularly for the two-argument properties, where things would otherwise go badly wrong.

.bool_set:N
.bool_set:c
.bool_gset:N
.bool_gset:c

One function for this.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:N } #1
  { \__keys_bool_set:Nn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_tl .bool_set:c } #1
  { \__keys_bool_set:cn {#1} { } }
\cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:N } #1
  { \__keys_bool_set:Nn #1 { g } }
\cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset:c } #1
  { \__keys_bool_set:cn {#1} { g } }
```

(*End definition for `.bool_set:N` and `.bool_set:c`. These functions are documented on page 152.*)

.bool_set_inverse:N
.bool_set_inverse:c
.bool_gset_inverse:N
.bool_gset_inverse:c

One function for this.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:N } #1
  { \__keys_bool_set_inverse:Nn #1 { } }
\cs_new_protected:cpn { \c__keys_props_root_tl .bool_set_inverse:c } #1
  { \__keys_bool_set_inverse:cn {#1} { } }
\cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:N } #1
  { \__keys_bool_set_inverse:Nn #1 { g } }
\cs_new_protected:cpn { \c__keys_props_root_tl .bool_gset_inverse:c } #1
  { \__keys_bool_set_inverse:cn {#1} { g } }
```

(*End definition for `.bool_set_inverse:N` and `.bool_set_inverse:c`. These functions are documented on page 152.*)

.choice:

Making a choice is handled internally, as it is also needed by `.generate_choices:n`.

```
\cs_new_protected_nopar:cpn { \c__keys_props_root_tl .choice: }
  { \__keys_choice_make: }
```

(*End definition for `.choice:`. This function is documented on page 152.*)

.choices:nn
.choices:Vn
.choices:on
.choices:xn

For auto-generation of a series of mutually-exclusive choices. Here, `#1` will consist of two separate arguments, hence the slightly odd-looking implementation.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .choices:nn } #1
  { \__keys_choices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_tl .choices:Vn } #1
  { \exp_args:NV \__keys_choices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_tl .choices:on } #1
  { \exp_args:No \__keys_choices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_tl .choices:xn } #1
  { \exp_args:Nx \__keys_choices_make:nn #1 }
```

(*End definition for `.choices:nn` and others. These functions are documented on page 152.*)

.code:n

Creating code is simply a case of passing through to the underlying set function.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .code:n } #1
  { \__keys_cmd_set:nn { \l_keys_path_tl } {#1} }
```

(*End definition for .code:n. This function is documented on page 152.*)

.clist_set:N
.clist_set:c
.clist_gset:N
.clist_gset:c

```
\cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:N } #1
  { \__keys_variable_set:NnnN #1 { clist } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:c } #1
  { \__keys_variable_set:cnnN {#1} { clist } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:N } #1
  { \__keys_variable_set:NnnN #1 { clist } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:c } #1
  { \__keys_variable_set:cnnN {#1} { clist } { g } n }
```

(*End definition for .clist_set:N and .clist_set:c. These functions are documented on page 152.*)

.default:n
.default:V
.default:o
.default:x

Expansion is left to the internal functions.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .default:n } #1
  { \__keys_default_set:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_tl .default:V } #1
  { \exp_args:NV \__keys_default_set:n #1 }
\cs_new_protected:cpn { \c__keys_props_root_tl .default:o } #1
  { \exp_args:No \__keys_default_set:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_tl .default:x } #1
  { \exp_args:Nx \__keys_default_set:n {#1} }
```

(*End definition for .default:n and others. These functions are documented on page 153.*)

.dim_set:N
.dim_set:c
.dim_gset:N
.dim_gset:c

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:N } #1
  { \__keys_variable_set:NnnN #1 { dim } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:c } #1
  { \__keys_variable_set:cnnN {#1} { dim } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:N } #1
  { \__keys_variable_set:NnnN #1 { dim } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1
  { \__keys_variable_set:cnnN {#1} { dim } { g } n }
```

(*End definition for .dim_set:N and .dim_set:c. These functions are documented on page 153.*)

.fp_set:N
.fp_set:c
.fp_gset:N
.fp_gset:c

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:N } #1
  { \__keys_variable_set:NnnN #1 { fp } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:c } #1
  { \__keys_variable_set:cnnN {#1} { fp } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:N } #1
  { \__keys_variable_set:NnnN #1 { fp } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1
  { \__keys_variable_set:cnnN {#1} { fp } { g } n }
```

(*End definition for .fp_set:N and .fp_set:c. These functions are documented on page 153.*)

.groups:n A single property to create groups of keys.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .groups:n } #1
  { \__keys_groups_set:n {#1} }
```

(*End definition for* `.groups:n`*. This function is documented on page 153.*)

.initial:n
.initial:V
.initial:o
.initial:x

The standard hand-off approach.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .initial:n } #1
  { \__keys_initialise:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_tl .initial:V } #1
  { \exp_args:NV \__keys_initialise:n #1 }
\cs_new_protected:cpn { \c__keys_props_root_tl .initial:o } #1
  { \exp_args:No \__keys_initialise:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_tl .initial:x } #1
  { \exp_args:Nx \__keys_initialise:n {#1} }
```

(*End definition for* `.initial:n` *and others. These functions are documented on page 153.*)

.int_set:N
.int_set:c
.int_gset:N
.int_gset:c

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .int_set:N } #1
  { \__keys_variable_set:NnnN #1 { int } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .int_set:c } #1
  { \__keys_variable_set:cnnN {#1} { int } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:N } #1
  { \__keys_variable_set:NnnN #1 { int } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:c } #1
  { \__keys_variable_set:cnnN {#1} { int } { g } n }
```

(*End definition for* `.int_set:N` *and* `.int_set:c`*. These functions are documented on page 153.*)

.meta:n Making a meta is handled internally.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .meta:n } #1
  { \__keys_meta_make:n {#1} }
```

(*End definition for* `.meta:n`*. This function is documented on page 154.*)

.meta:nn Meta with path: potentially lots of variants, but for the moment no so many defined.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .meta:nn } #1
  { \__keys_meta_make:nn #1 }
```

(*End definition for* `.meta:nn`*. This function is documented on page 154.*)

.multichoice:
.multichoices:nn
.multichoices:Vn
.multichoices:on
.multichoices:xn

The same idea as `.choice:` and `.choices:nn`, but where more than one choice is allowed.

```
\cs_new_protected_nopar:cpn { \c__keys_props_root_tl .multichoice: }
  { \__keys_multichoice_make: }
\cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:nn } #1
  { \__keys_multichoices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:Vn } #1
  { \exp_args:NV \__keys_multichoices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:on } #1
  { \exp_args:No \__keys_multichoices_make:nn #1 }
\cs_new_protected:cpn { \c__keys_props_root_tl .multichoices:xn } #1
  { \exp_args:Nx \__keys_multichoices_make:nn #1 }
```

(*End definition for* `.multichoice:`. *This function is documented on page 154.*)

`.skip_set:N` `.skip_set:c` `.skip_gset:N` `.skip_gset:c`

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:N } #1
  { \__keys_variable_set:NnnN #1 { skip } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .skip_set:c } #1
  { \__keys_variable_set:cnnN {#1} { skip } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:N } #1
  { \__keys_variable_set:NnnN #1 { skip } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .skip_gset:c } #1
  { \__keys_variable_set:cnnN {#1} { skip } { g } n }
```

(*End definition for* `.skip_set:N` *and* `.skip_set:c`. *These functions are documented on page 154.*)

`.tl_set:N` `.tl_set:c` `.tl_gset:N` `.tl_gset:c` `.tl_set_x:N` `.tl_set_x:c` `.tl_gset_x:N` `.tl_gset_x:c`

Setting a variable is very easy: just pass the data along.

```
\cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:N } #1
  { \__keys_variable_set:NnnN #1 { tl } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .tl_set:c } #1
  { \__keys_variable_set:cnnN {#1} { tl } { } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:N } #1
  { \__keys_variable_set:NnnN #1 { tl } { } x }
\cs_new_protected:cpn { \c__keys_props_root_tl .tl_set_x:c } #1
  { \__keys_variable_set:cnnN {#1} { tl } { } x }
\cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:N } #1
  { \__keys_variable_set:NnnN #1 { tl } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset:c } #1
  { \__keys_variable_set:cnnN {#1} { tl } { g } n }
\cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:N } #1
  { \__keys_variable_set:NnnN #1 { tl } { g } x }
\cs_new_protected:cpn { \c__keys_props_root_tl .tl_gset_x:c } #1
  { \__keys_variable_set:cnnN {#1} { tl } { g } x }
```

(*End definition for* `.tl_set:N` *and* `.tl_set:c`. *These functions are documented on page 154.*)

`.value_forbidden:` `.value_required:`

These are very similar, so both call the same function.

```
\cs_new_protected_nopar:cpn { \c__keys_props_root_tl .value_forbidden: }
  { \__keys_value_requirement:n { forbidden } }
\cs_new_protected_nopar:cpn { \c__keys_props_root_tl .value_required: }
  { \__keys_value_requirement:n { required } }
```

(*End definition for* `.value_forbidden:`. *This function is documented on page 154.*)

18.6 Setting keys

`\keys_set:nn` `\keys_set:nV` `\keys_set:nv` `\keys_set:no` `\__keys_set:nnn` `\__keys_set:onn`

A simple wrapper again.

```
\cs_new_protected_nopar:Npn \keys_set:nn
  { \__keys_set:onn { \l__keys_module_tl } }
\cs_new_protected:Npn \__keys_set:nnn #1#2#3
  {
    \tl_set:Nx \l__keys_module_tl { \tl_to_str:n {#2} }
    \keyval_parse:NNn \__keys_set_elt:n \__keys_set_elt:nn {#3}
```

```
    \tl_set:Nn \l__keys_module_tl {#1}
  }
\cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
\cs_generate_variant:Nn \__keys_set:nnn { o }
```

(*End definition for* `\keys_set:nn` *and others. These functions are documented on page* **??**.)

`\keys_set_known:nnN`
`\keys_set_known:nVN`
`\keys_set_known:nvN`
`\keys_set_known:noN`
`\__keys_set_known:nnnN`
`\__keys_set_known:onnN`
`\keys_set_known:nn`
`\keys_set_known:nV`
`\keys_set_known:nv`
`\keys_set_known:no`

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!

```
\cs_new_protected_nopar:Npn \keys_set_known:nnN
  { \__keys_set_known:onnN \l__keys_unused_clist }
\cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
\cs_new_protected:Npn \__keys_set_known:nnnN #1#2#3#4
  {
    \clist_clear:N \l__keys_unused_clist
    \keys_set_known:nn {#2} {#3}
    \tl_set:Nx #4 { \exp_not:o { \l__keys_unused_clist } }
    \tl_set:Nn \l__keys_unused_clist {#1}
  }
\cs_generate_variant:Nn \__keys_set_known:nnnN { o }
\cs_new_protected:Npn \keys_set_known:nn #1#2
  {
    \bool_set_true:N \l__keys_only_known_bool
    \keys_set:nn {#1} {#2}
    \bool_set_false:N \l__keys_only_known_bool
  }
\cs_generate_variant:Nn \keys_set_known:nn { nV , nv , no }
```

(*End definition for* `\keys_set_known:nnN` *and others. These functions are documented on page* **??**.)

`\keys_set_filter:nnnN`
`\keys_set_filter:nnVN`
`\keys_set_filter:nnvN\keys_set_filter:nnoN`
`\__keys_set_filter:nnnnN`
`\__keys_set_filter:onnnN`
`\keys_set_filter:nnn`
`\keys_set_filter:nnV`
`\keys_set_filter:nnv\keys_set_filter:nno`
`\keys_set_groups:nnn`
`\keys_set_groups:nnV`
`\keys_set_groups:nnv\keys_set_groups:nno`

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.

```
\cs_new_protected_nopar:Npn \keys_set_filter:nnnN
  {  \__keys_set_filter:onnnN \l__keys_unused_clist }
\cs_generate_variant:Nn \keys_set_filter:nnnN { nnV , nnv , nno }
\cs_new_protected:Npn \__keys_set_filter:nnnnN #1#2#3#4#5
  {
    \clist_clear:N \l__keys_unused_clist
    \keys_set_filter:nnn {#2} {#3} {#4}
    \tl_set:Nx #5 { \exp_not:o { \l__keys_unused_clist } }
    \tl_set:Nn \l__keys_unused_clist {#1}
  }
\cs_generate_variant:Nn \__keys_set_filter:nnnnN { o }
\cs_new_protected:Npn \keys_set_filter:nnn #1#2#3
  {
    \bool_set_true:N \l__keys_selective_bool
    \bool_set_true:N \l__keys_filtered_bool
    \seq_set_from_clist:Nn \l__keys_selective_seq {#2}
    \keys_set:nn {#1} {#3}
```

```
    \bool_set_false:N \l__keys_selective_bool
  }
\cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }
\cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
  {
    \bool_set_true:N \l__keys_selective_bool
    \bool_set_false:N \l__keys_filtered_bool
    \seq_set_from_clist:Nn \l__keys_selective_seq {#2}
    \keys_set:nn {#1} {#3}
    \bool_set_false:N \l__keys_selective_bool
  }
\cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }
```

(*End definition for* `\keys_set_filter:nnnN`, `\keys_set_filter:nnVN`, *and* `\keys_set_filter:nnvN\keys_set_filter:nnoN`. *These functions are documented on page* **??**.)

`\__keys_set_elt:n`
`\__keys_set_elt:nn`
`\__keys_set_elt_aux:nn`
`\__keys_set_elt_aux:`
`\__keys_set_elt_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.

```
\cs_new_protected:Npn \__keys_set_elt:n #1
  {
    \bool_set_true:N \l__keys_no_value_bool
    \__keys_set_elt_aux:nn {#1} { }
  }
\cs_new_protected:Npn \__keys_set_elt:nn #1#2
  {
    \bool_set_false:N \l__keys_no_value_bool
    \__keys_set_elt_aux:nn {#1} {#2}
  }
\cs_new_protected:Npn \__keys_set_elt_aux:nn #1#2
  {
    \tl_set:Nx \l_keys_key_tl { \tl_to_str:n {#1} }
    \tl_set:Nx \l_keys_path_tl { \l__keys_module_tl / \l_keys_key_tl }
    \__keys_value_or_default:n {#2}
    \bool_if:NTF \l__keys_selective_bool
      { \__keys_set_elt_selective: }
      { \__keys_set_elt_aux: }
  }
\cs_new_protected_nopar:Npn \__keys_set_elt_aux:
  {
    \bool_if:nTF
      {
        \__keys_if_value_p:n { required } &&
        \l__keys_no_value_bool
      }
      {
        \__msg_kernel_error:nnx { kernel } { value-required }
          { \l_keys_path_tl }
      }
      {
```

```
        \bool_if:nTF
          {
              \__keys_if_value_p:n { forbidden } &&
            ! \l__keys_no_value_bool
          }
          {
            \__msg_kernel_error:nnxx { kernel } { value-forbidden }
              { \l_keys_path_tl } { \l_keys_value_tl }
          }
          { \__keys_execute: }
      }
  }
```

If selective setting is active, there are a number of possible sub-cases to consider. The key name may not be known at all or if it is, it may not have any groups assigned. There is then the question of whether the selection is opt-in or opt-out.

```
\cs_new_protected_nopar:Npn \__keys_set_elt_selective:
  {
    \prop_if_exist:cTF { \c__keys_info_root_tl \l_keys_path_tl }
      {
        \prop_get:cnNTF { \c__keys_info_root_tl \l_keys_path_tl }
          { groups } \l__keys_groups_clist
          { \__keys_check_groups: }
          {
            \bool_if:NTF \l__keys_filtered_bool
              { \__keys_set_elt_aux: }
              { \__keys_store_unused: }
          }
      }
      {
        \bool_if:NTF \l__keys_filtered_bool
          { \__keys_set_elt_aux: }
          { \__keys_store_unused: }
      }
  }
```

In the case where selective setting requires a comparison of the list of groups which apply to a key with the list of those which have been set active. That requires two mappings, and again a different outcome depending on whether opt-in or opt-out is set.

```
\cs_new_protected_nopar:Npn \__keys_check_groups:
  {
    \bool_set_false:N \l__keys_tmp_bool
    \seq_map_inline:Nn \l__keys_selective_seq
      {
        \clist_map_inline:Nn \l__keys_groups_clist
          {
            \str_if_eq:nnT {##1} {####1}
              {
                \bool_set_true:N \l__keys_tmp_bool
                \clist_map_break:n { \seq_map_break: }
```

```
            }
          }
      }
    \bool_if:NTF \l__keys_tmp_bool
      {
        \bool_if:NTF \l__keys_filtered_bool
          { \__keys_store_unused: }
          { \__keys_set_elt_aux: }
      }
      {
        \bool_if:NTF \l__keys_filtered_bool
          { \__keys_set_elt_aux: }
          { \__keys_store_unused: }
      }
  }
```

(*End definition for* `\__keys_set_elt:n` *and* `\__keys_set_elt:nn`.)

`\__keys_value_or_default:n`

If a value is given, return it as #1, otherwise send a default if available.

```
\cs_new_protected:Npn \__keys_value_or_default:n #1
  {
    \bool_if:NTF \l__keys_no_value_bool
      {
        \prop_get:cnNF { \c__keys_info_root_tl \l_keys_path_tl }
          { default } \l_keys_value_tl
          { \tl_clear:N \l_keys_value_tl }
      }
      { \tl_set:Nn \l_keys_value_tl {#1} }
  }
```

(*End definition for* `\__keys_value_or_default:n`.)

`\__keys_if_value_p:n`

To test if a value is required or forbidden. A simple check for the existence of the appropriate marker.

```
\prg_new_conditional:Npnn \__keys_if_value:n #1 { p }
  {
    \prop_if_exist:cTF { \c__keys_info_root_tl \l_keys_path_tl }
      {
        \prop_if_in:cnTF { \c__keys_info_root_tl \l_keys_path_tl } {#1}
          { \prg_return_true: }
          { \prg_return_false: }
      }
      { \prg_return_false: }
  }
```

(*End definition for* `\__keys_if_value_p: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.

```
\cs_new_protected_nopar:Npn \__keys_execute:
```

```
  { \__keys_execute:nn { \l_keys_path_tl } { \__keys_execute_unknown: } }
\cs_new_protected_nopar:Npn \__keys_execute_unknown:
  {
    \bool_if:NTF \l__keys_only_known_bool
      { \__keys_store_unused: }
      {
        \__keys_execute:nn { \l__keys_module_tl / unknown }
          {
            \__msg_kernel_error:nnxx { kernel } { key-unknown }
              { \l_keys_path_tl } { \l__keys_module_tl }
          }
      }
  }
\cs_new:Npn \__keys_execute:nn #1#2
  {
    \cs_if_exist:cTF { \c__keys_code_root_tl #1 }
      {
        \exp_args:Nc \exp_args:No { \c__keys_code_root_tl #1 }
          \l_keys_value_tl
      }
      {#2}
  }
\cs_new_protected_nopar:Npn \__keys_store_unused:
  {
    \clist_put_right:Nx \l__keys_unused_clist
      {
        \exp_not:o \l_keys_key_tl
        \bool_if:NF \l__keys_no_value_bool
          { = { \exp_not:o \l_keys_value_tl } }
      }
  }
```

(*End definition for* `\__keys_execute:`.)

`\__keys_choice_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.

```
\cs_new:Npn \__keys_choice_find:n #1
  {
    \__keys_execute:nn { \l_keys_path_tl / \tl_to_str:n {#1} }
      { \__keys_execute:nn { \l_keys_path_tl / unknown } { } }
  }
```

(*End definition for* `\__keys_choice_find:n`.)

18.7 Utilities

`\keys_if_exist_p:nn`
`\keys_if_exist:nnTF`

A utility for others to see if a key exists.

```
\prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
  {
    \cs_if_exist:cTF { \c__keys_code_root_tl #1 / #2 }
```

```
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End definition for* `\keys_if_exist:nn`. *These functions are documented on page* 160.)

`\keys_if_choice_exist_p:nnn`
`\keys_if_choice_exist:nnnTF`

Just an alternative view on `\keys_if_exist:nn(TF)`.

```
\prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3 { p , T , F , TF }
  {
    \cs_if_exist:cTF { \c__keys_code_root_tl #1 / #2 / #3 }
      { \prg_return_true: }
      { \prg_return_false: }
  }
```

(*End definition for* `\keys_if_choice_exist:nnn`. *These functions are documented on page* 160.)

`\keys_show:nn`

Showing a key is just a question of using the correct name.

```
\cs_new_protected:Npn \keys_show:nn #1#2
  { \cs_show:c { \c__keys_code_root_tl #1 / \tl_to_str:n {#2} } }
```

(*End definition for* `\keys_show:nn`. *This function is documented on page* 160.)

18.8 Messages

For when there is a need to complain.

```
\__msg_kernel_new:nnnn { kernel } { boolean-values-only }
  { Key~'#1'~accepts~boolean~values~only. }
  { The~key~'#1'~only~accepts~the~values~'true'~and~'false'. }
\__msg_kernel_new:nnnn { kernel } { choice-unknown }
  { Choice~'#2'~unknown~for~key~'#1'. }
  {
    The~key~'#1'~takes~a~limited~number~of~values.\\
    The~input~given,~'#2',~is~not~on~the~list~accepted.
  }
\__msg_kernel_new:nnnn { kernel } { key-choice-unknown }
  { Key~'#1'~accepts~only~a~fixed~set~of~choices. }
  {
    The~key~'#1'~only~accepts~predefined~values,~and~'#2'~is~not~one~of~these.
  }
\__msg_kernel_new:nnnn { kernel } { key-no-property }
  { No~property~given~in~definition~of~key~'#1'. }
  {
    \c_msg_coding_error_text_tl
    Inside~\keys_define:nn  each~key~name~
    needs~a~property:  \\ \\
    \iow_indent:n { #1 .<property> } \\ \\
    LaTeX~did~not~find~a~'.'~to~indicate~the~start~of~a~property.
  }
\__msg_kernel_new:nnnn { kernel } { key-unknown }
  { The~key~'#1'~is~unknown~and~is~being~ignored. }
  {
    The~module~'#2'~does~not~have~a~key~called~#1'.\\
```

```
    Check~that~you~have~spelled~the~key~name~correctly.
  }
\__msg_kernel_new:nnnn { kernel } { property-requires-value }
  { The~property~'#1'~requires~a~value. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~was~asked~to~set~property~'#1'~for~key~'#2'.\\
    No~value~was~given~for~the~property,~and~one~is~required.
  }
\__msg_kernel_new:nnnn { kernel } { property-unknown }
  { The~key~property~'#1'~is~unknown. }
  {
    \c_msg_coding_error_text_tl
    LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
    this~property~is~not~defined.
  }
\__msg_kernel_new:nnnn { kernel } { value-forbidden }
  { The~key~'#1'~does~not~taken~a~value. }
  {
    The~key~'#1'~should~be~given~without~a~value.\\
    LaTeX~will~ignore~the~given~value~'#2'.
  }
\__msg_kernel_new:nnnn { kernel } { value-required }
  { The~key~'#1'~requires~a~value. }
  {
    The~key~'#1'~must~have~a~value.\\
    No~value~was~present:~the~key~will~be~ignored.
  }
```

18.9 Deprecated functions

__keys_choice_code_store:n
__keys_choice_code_store:x
.choice_code:n
.choice_code:x
__keys_choices_generate:n
__keys_choices_generate_aux:n
.generate_choices:n

Deprecated on 2013-07-09.

```
\cs_new_protected:Npn \__keys_choice_code_store:n #1
  {
    \cs_if_exist:cF
      { \c__keys_info_root_tl \l_keys_path_tl .choice~code }
      {
        \tl_new:c
          { \c__keys_info_root_tl \l_keys_path_tl .choice~code }
      }
    \tl_set:cn { \c__keys_info_root_tl \l_keys_path_tl .choice~code }
      {#1}
  }
\cs_generate_variant:Nn \__keys_choice_code_store:n { x }
\cs_new_protected:cpn { \c__keys_props_root_tl .choice_code:n } #1
  { \__keys_choice_code_store:n {#1} }
\cs_new_protected:cpn { \c__keys_props_root_tl .choice_code:x } #1
  { \__keys_choice_code_store:x {#1} }
\cs_new_protected:Npn \__keys_choices_generate:n #1
```

```
  {
    \cs_if_exist:cTF
      { \c__keys_info_root_tl \l_keys_path_tl .choice~code }
      {
        \__keys_choice_make:
        \int_zero:N \l_keys_choice_int
        \clist_map_function:nN {#1} \__keys_choices_generate_aux:n
      }
      {
        \__msg_kernel_error:nnx { kernel }
          { generate-choices-before-code } { \l_keys_path_tl }
      }
  }
\cs_new_protected:Npn \__keys_choices_generate_aux:n #1
  {
    \int_incr:N \l_keys_choice_int
    \__keys_cmd_set:nx { \l_keys_path_tl / #1 }
      {
        \tl_set:Nn \exp_not:N \l_keys_choice_tl {#1}
        \int_set:Nn \exp_not:N \l_keys_choice_int
          { \int_use:N \l_keys_choice_int }
        \exp_not:v
          { \c__keys_info_root_tl \l_keys_path_tl .choice~code }
      }
  }
\__msg_kernel_new:nnnn { kernel } { generate-choices-before-code }
  { No~code~available~to~generate~choices~for~key~'#1'. }
  {
    \c_msg_coding_error_text_tl
    Before~using~.generate_choices:n~the~code~should~be~defined~
    with~'.choice_code:n'~or~'.choice_code:x'.
  }
\cs_new_protected:cpn { \c__keys_props_root_tl .generate_choices:n } #1
  { \__keys_choices_generate:n {#1} }
```

(*End definition for* `\__keys_choice_code_store:n` *and* `\__keys_choice_code_store:x`*.*)

⟨/initex | package⟩

19 l3file implementation

The following test files are used for this code: m3file001.

⟨*initex | package⟩

⟨@@=file⟩

19.1 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 package mode the current file name is collected from $\LaTeX 2_\varepsilon$.

```
\tl_new:N \g_file_current_name_tl
⟨*initex⟩
\tex_everyjob:D \exp_after:wN
  {
    \tex_the:D \tex_everyjob:D
    \tl_gset:Nx \g_file_current_name_tl { \tex_jobname:D }
  }
⟨/initex⟩
⟨*package⟩
\tl_gset_eq:NN \g_file_current_name_tl \@currname
⟨/package⟩
```

(*End definition for* `\g_file_current_name_tl`*. This variable is documented on page 162.*)

`\g__file_stack_seq` The input list of files is stored as a sequence stack.

```
\seq_new:N \g__file_stack_seq
```

(*End definition for* `\g__file_stack_seq`*. This variable is documented on page* ??.)

`\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`.

```
\seq_new:N \g__file_record_seq
⟨*initex⟩
\tex_everyjob:D \exp_after:wN
  {
    \tex_the:D \tex_everyjob:D
    \seq_gput_right:NV \g__file_record_seq \g_file_current_name_tl
  }
⟨/initex⟩
```

(*End definition for* `\g__file_record_seq`*. This variable is documented on page* ??.)

`\l__file_internal_tl` Used as a short-term scratch variable. It may be possible to reuse `\l__file_internal_name_tl` there.

```
\tl_new:N \l__file_internal_tl
```

(*End definition for* `\l__file_internal_tl`*. This variable is documented on page* ??.)

`\l__file_internal_name_tl` Used to return the fully-qualified name of a file.

```
\tl_new:N \l__file_internal_name_tl
```

(*End definition for* `\l__file_internal_name_tl`*. This variable is documented on page 168.*)

`\l__file_search_path_seq` The current search path.

```
\seq_new:N \l__file_search_path_seq
```

(*End definition for* `\l__file_search_path_seq`*. This variable is documented on page* ??.)

\l__file_saved_search_path_seq The current search path has to be saved for package use.

```
⟨*package⟩
\seq_new:N \l__file_saved_search_path_seq
⟨/package⟩
```

(*End definition for* `\l__file_saved_search_path_seq`. *This variable is documented on page* **??**.)

\l__file_internal_seq Scratch space for comma list conversion in package mode.

```
⟨*package⟩
\seq_new:N \l__file_internal_seq
⟨/package⟩
```

(*End definition for* `\l__file_internal_seq`. *This variable is documented on page* **??**.)

__file_name_sanitize:nn For converting a token list to a string where active characters are treated as strings from the start.

```
\cs_new_protected:Npn \__file_name_sanitize:nn #1#2
  {
    \group_begin:
      \seq_map_inline:Nn \l_char_active_seq
        { \cs_set_nopar:Npx ##1 { \token_to_str:N ##1 } }
      \tl_set:Nx \l__file_internal_name_tl {#1}
      \tl_set:Nx \l__file_internal_name_tl
        { \tl_to_str:N \l__file_internal_name_tl }
      \tl_if_in:NnT \l__file_internal_name_tl { ~ }
        {
          \__msg_kernel_error:nnx { kernel } { space-in-file-name }
            { \l__file_internal_name_tl }
          \tl_remove_all:Nn \l__file_internal_name_tl { ~ }
        }
      \use:x
        {
          \group_end:
          \exp_not:n {#2} { \l__file_internal_name_tl }
        }
  }
```

(*End definition for* `\__file_name_sanitize:nn`.)

\file_add_path:nN
__file_add_path:nN
__file_add_path_search:nN

The way to test if a file exists is to try to open it: if it does not exist then TeX will report end-of-file. For files which are in the current directory, this is straight-forward. 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.

```
\cs_new_protected:Npn \file_add_path:nN #1
  { \__file_name_sanitize:nn {#1} { \__file_add_path:nN } }
\cs_new_protected:Npn \__file_add_path:nN #1#2
  {
    \__ior_open:Nn \g__file_internal_ior {#1}
    \ior_if_eof:NTF \g__file_internal_ior
      { \__file_add_path_search:nN {#1} #2 }
```

```
      { \tl_set:Nn #2 {#1} }
    \ior_close:N \g__file_internal_ior
  }
\cs_new_protected:Npn \__file_add_path_search:nN #1#2
  {
    \tl_set:Nn #2 { \q_no_value  }
⟨*package⟩
    \cs_if_exist:NT \input@path
      {
        \seq_set_eq:NN \l__file_saved_search_path_seq \l__file_search_path_seq
        \seq_set_split:NnV \l__file_internal_seq { , } \input@path
        \seq_concat:NNN \l__file_search_path_seq
          \l__file_search_path_seq \l__file_internal_seq
      }
⟨/package⟩
    \seq_map_inline:Nn \l__file_search_path_seq
      {
        \__ior_open:Nn \g__file_internal_ior { ##1 #1 }
        \ior_if_eof:NF \g__file_internal_ior
          {
            \tl_set:Nx #2 { ##1 #1 }
            \seq_map_break:
          }
      }
⟨*package⟩
    \cs_if_exist:NT \input@path
      { \seq_set_eq:NN \l__file_search_path_seq \l__file_saved_search_path_seq }
⟨/package⟩
  }
```

(*End definition for* `\file_add_path:nN`*. This function is documented on page 162.*)

\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`.

```
\prg_new_protected_conditional:Npnn \file_if_exist:n #1 { T , F , TF }
  {
    \file_add_path:nN {#1} \l__file_internal_name_tl
    \quark_if_no_value:NTF \l__file_internal_name_tl
      { \prg_return_false: }
      { \prg_return_true: }
  }
```

(*End definition for* `\file_if_exist:nTF`*. This function is documented on page 162.*)

\file_input:n
__file_if_exist:nT
__file_input:n__file_input:V
__file_input_aux:n
__file_input_aux:o

Loading a file is done in a safe way, checking first that the file exists and loading only if it does. Push the file name on the `\g__file_stack_seq`, and add it to the file list, either `\g__file_record_seq`, or `\@filelist` in package mode.

```
\cs_new_protected:Npn \file_input:n #1
  {
    \__file_if_exist:nT {#1}
```

```
      { \__file_input:V \l__file_internal_name_tl }
  }
```

This code is spun out as a separate function so it is available for other kernel file operations which have the same logic.

```
\cs_new_protected:Npn \__file_if_exist:nT #1#2
  {
    \file_add_path:nN {#1} \l__file_internal_name_tl
    \quark_if_no_value:NTF \l__file_internal_name_tl
      {
        \__file_name_sanitize:nn {#1}
          { \__msg_kernel_error:nnx { kernel } { file-not-found } }
      }
      { #2 }
  }
\cs_new_protected:Npn \__file_input:n #1
  {
    \tl_if_in:nnTF {#1} { . }
      { \__file_input_aux:n {#1} }
      { \__file_input_aux:o { \tl_to_str:n { #1 . tex } } }
  }
\cs_generate_variant:Nn \__file_input:n { V }
\cs_new_protected:Npn \__file_input_aux:n #1
  {
⟨*initex⟩
    \seq_gput_right:Nn \g__file_record_seq {#1}
⟨/initex⟩
⟨*package⟩
    \clist_if_exist:NTF \@filelist
      { \@addtofilelist {#1} }
      { \seq_gput_right:Nn \g__file_record_seq {#1} }
⟨/package⟩
    \seq_gpush:No \g__file_stack_seq \g_file_current_name_tl
    \tl_gset:Nn \g_file_current_name_tl {#1}
    \tex_input:D #1 \c_space_tl
    \seq_gpop:NN \g__file_stack_seq \l__file_internal_tl
    \tl_gset_eq:NN \g_file_current_name_tl \l__file_internal_tl
  }
\cs_generate_variant:Nn \__file_input_aux:n { o }
```

(*End definition for* `\file_input:n`. *This function is documented on page* ??.)

\file_path_include:n
\file_path_remove:n
__file_path_include:n

Wrapper functions to manage the search path.

```
\cs_new_protected:Npn \file_path_include:n #1
  { \__file_name_sanitize:nn {#1} { \__file_path_include:n } }
\cs_new_protected:Npn \__file_path_include:n #1
  {
    \seq_if_in:NnF \l__file_search_path_seq {#1}
      { \seq_put_right:Nn \l__file_search_path_seq {#1} }
  }
\cs_new_protected:Npn \file_path_remove:n #1
```

```
  {
    \__file_name_sanitize:nn {#1}
      { \seq_remove_all:Nn \l__file_search_path_seq }
  }
```

(*End definition for* `\file_path_include:n`. *This function is documented on page 163.*)

`\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 it into account (we capture it `\AtBeginDocument` into `\g__file_record_seq`), turning each file name into a string.

```
\cs_new_protected_nopar:Npn \file_list:
  {
    \seq_set_eq:NN \l__file_internal_seq \g__file_record_seq
⟨*package⟩
    \clist_if_exist:NT \@filelist
      {
        \clist_map_inline:Nn \@filelist
          {
            \seq_put_right:No \l__file_internal_seq
              { \tl_to_str:n {##1} }
          }
      }
⟨/package⟩
    \seq_remove_duplicates:N \l__file_internal_seq
    \iow_log:n { *~File~List~* }
    \seq_map_inline:Nn \l__file_internal_seq { \iow_log:n {##1} }
    \iow_log:n { ************* }
  }
```

(*End definition for* `\file_list:`. *This function is documented on page 163.*)

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`.

```
⟨*package⟩
\AtBeginDocument
  {
    \clist_map_inline:Nn \@filelist
      { \seq_gput_right:No \g__file_record_seq { \tl_to_str:n {#1} } }
  }
⟨/package⟩
```

19.2 Input operations

```
⟨@@=ior⟩
```

19.2.1 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.

```
\cs_new_eq:NN \c_term_ior \c_sixteen
```

(*End definition for* `\c_term_ior`. *This variable is documented on page 168.*)

`\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 $\LaTeX 2_\varepsilon$ as they are needed (initially none are needed), so the starting point varies!

```
\seq_new:N \g__ior_streams_seq
⟨*initex⟩
\seq_gset_split:Nnn \g__ior_streams_seq { , }
  { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 }
⟨/initex⟩
```

(*End definition for* `\g__ior_streams_seq`*. This variable is documented on page* **??**.)

`\l__ior_stream_tl`

Used to recover the raw stream number from the stack.

```
\tl_new:N \l__ior_stream_tl
```

(*End definition for* `\l__ior_stream_tl`*. This variable is documented on page* **??**.)

`\g__ior_streams_prop`

The name of the file attached to each stream is tracked in a property list.

```
\prop_new:N \g__ior_streams_prop
⟨*package⟩
\prop_gput:Nnn \g__ior_streams_prop { 0 } { LaTeX2e~reserved }
⟨/package⟩
```

(*End definition for* `\g__ior_streams_prop`*. This variable is documented on page* **??**.)

19.2.2 Stream management

`\ior_new:N`
`\ior_new:c`

Reserving a new stream is done by defining the name as equal to using the terminal.

```
\cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c_term_ior }
\cs_generate_variant:Nn \ior_new:N { c }
```

(*End definition for* `\ior_new:N` *and* `\ior_new:c`*. These functions are documented on page* **??**.)

`\ior_open:Nn`
`\ior_open:cn`
`\__ior_open_aux:Nn`

Opening an input stream requires a bit of pre-processing. The file name is sanitized to deal with active characters, before an auxiliary adds a path and checks that the file really exists. If those two tests pass, then pass the information on to the lower-level function which deals with streams.

```
\cs_new_protected:Npn \ior_open:Nn #1#2
  { \__file_name_sanitize:nn {#2} { \__ior_open_aux:Nn #1 } }
\cs_generate_variant:Nn \ior_open:Nn { c }
\cs_new_protected:Npn \__ior_open_aux:Nn #1#2
  {
    \file_add_path:nN {#2} \l__file_internal_name_tl
    \quark_if_no_value:NTF \l__file_internal_name_tl
      { \__msg_kernel_error:nnx { kernel } { file-not-found } {#2} }
      { \__ior_open:No #1 \l__file_internal_name_tl }
  }
```

(*End definition for* `\ior_open:Nn` *and* `\ior_open:cn`*. These functions are documented on page* **??**.)

\ior_open:NnTF
\ior_open:cnTF
__ior_open_aux:NnTF

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.

```
\prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
  { \__file_name_sanitize:nn {#2} { \__ior_open_aux:NnTF #1 } }
\cs_generate_variant:Nn \ior_open:NnT  { c }
\cs_generate_variant:Nn \ior_open:NnF  { c }
\cs_generate_variant:Nn \ior_open:NnTF { c }
\cs_new_protected:Npn \__ior_open_aux:NnTF #1#2
  {
    \file_add_path:nN {#2} \l__file_internal_name_tl
    \quark_if_no_value:NTF \l__file_internal_name_tl
      { \prg_return_false: }
      {
        \__ior_open:No #1 \l__file_internal_name_tl
        \prg_return_true:
      }
  }
```

(*End definition for* \ior_open:NnTF *and* \ior_open:cnTF*. These functions are documented on page* **??**.)

__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 $\LaTeX2_\varepsilon$ for a new stream and use that number (after a bit of conversion).

```
\cs_new_protected:Npn \__ior_open:Nn #1#2
  {
    \ior_close:N #1
    \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
      { \__ior_open_stream:Nn #1 {#2} }
⟨*initex⟩
      { \__msg_kernel_fatal:nn { kernel } { input-streams-exhausted } }
⟨/initex⟩
⟨*package⟩
      {
        \cs:w newread \cs_end: #1
        \tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
        \__ior_open_stream:Nn #1 {#2}
      }
⟨/package⟩
  }
\cs_generate_variant:Nn \__ior_open:Nn { No }
\cs_new_protected:Npn \__ior_open_stream:Nn #1#2
  {
    \tex_global:D \tex_chardef:D #1 = \l__ior_stream_tl \scan_stop:
    \prop_gput:NVn \g__ior_streams_prop #1 {#2}
    \tex_openin:D #1 #2 \scan_stop:
  }
```

(*End definition for* __ior_open:Nn *and* __ior_open:No*.*)

\ior_close:N
\ior_close:c

Closing a stream means getting rid of it at the TeX level and removing from the various data structures. Unless the name passed is an invalid stream number (outside the range [0, 15]), it can be closed. On the other hand, it only gets added to the stack if it was not already there, to avoid duplicates building up.

```
\cs_new_protected:Npn \ior_close:N #1
  {
    \int_compare:nT { \c_minus_one < #1 < \c_sixteen }
      {
        \tex_closein:D #1
        \prop_gremove:NV \g__ior_streams_prop #1
        \seq_if_in:NVF \g__ior_streams_seq #1
          { \seq_gpush:NV \g__ior_streams_seq #1 }
        \cs_gset_eq:NN #1 \c_term_ior
      }
  }
\cs_generate_variant:Nn \ior_close:N { c }
```

(*End definition for* `\ior_close:N` *and* `\ior_close:c`*. These functions are documented on page* ??.)

\ior_list_streams:
__ior_list_streams:Nn

Show the property lists, but with some "pretty printing". See the l3msg module. If there are no open read streams, issue the message `show-no-stream`, and show an empty token list. If there are open read streams, format them with `\__msg_show_item_unbraced:nn`, and with the message `show-open-streams`.

```
\cs_new_protected_nopar:Npn \ior_list_streams:
  { \__ior_list_streams:Nn \g__ior_streams_prop { input } }
\cs_new_protected:Npn \__ior_list_streams:Nn #1#2
  {
    \__msg_term:nnn { LaTeX / kernel }
      { \prop_if_empty:NTF #1 { show-no-stream } { show-open-streams } }
      {#2}
    \__msg_show_variable:n
      { \prop_map_function:NN #1 \__msg_show_item_unbraced:nn }
  }
```

(*End definition for* `\ior_list_streams:`*. This function is documented on page* *164**.*)

19.2.3 Reading input

\if_eof:w

The primitive conditional

```
\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.

```
\prg_new_conditional:Nnn \ior_if_eof:N { p , T , F , TF }
  {
    \cs_if_exist:NTF #1
      {
        \if_int_compare:w #1 = \c_sixteen
          \prg_return_true:
        \else:
```

```
          \if_eof:w #1
            \prg_return_true:
          \else:
            \prg_return_false:
          \fi:
        \fi:
      }
      { \prg_return_true: }
  }
```

(*End definition for* `\ior_if_eof:N`. *These functions are documented on page 165.*)

`\ior_get:NN` And here we read from files.

```
\cs_new_protected:Npn \ior_get:NN #1#2
  { \tex_read:D #1 to #2 }
```

(*End definition for* `\ior_get:NN`. *This function is documented on page 164.*)

`\ior_get_str:NN` Reading as strings is a more complicated wrapper, as we wish to remove the endline character.

```
\cs_new_protected:Npn \ior_get_str:NN #1#2
  {
    \use:x
      {
        \int_set_eq:NN \tex_endlinechar:D \c_minus_one
        \exp_not:n { \etex_readline:D #1 to #2 }
        \int_set:Nn \tex_endlinechar:D { \int_use:N \tex_endlinechar:D }
      }
  }
```

(*End definition for* `\ior_get_str:NN`. *This function is documented on page 165.*)

`\g__file_internal_ior` Needed by the higher-level code, but cannot be created until here.

```
\ior_new:N \g__file_internal_ior
```

(*End definition for* `\g__file_internal_ior`. *This variable is documented on page* **??**.)

19.3 Output operations

```
⟨@@=iow⟩
```

There is a lot of similarity here to the input operations, at least for many of the basics. Thus quite a bit is copied from the earlier material with minor alterations.

19.3.1 Variables and constants

`\c_log_iow`
`\c_term_iow` Here we allocate two output streams for writing to the transcript file only (`\c_log_iow`) and to both the terminal and transcript file (`\c_term_iow`).

```
\cs_new_eq:NN \c_log_iow  \c_minus_one
\cs_new_eq:NN \c_term_iow \c_sixteen
```

(*End definition for* `\c_log_iow` *and* `\c_term_iow`. *These variables are documented on page 168.*)

\g__iow_streams_seq

A list of the currently-available input streams to be used as a stack. Things are done differently in format and package mode, so the starting point varies!

```
\seq_new:N \g__iow_streams_seq
⟨*initex⟩
\seq_gset_eq:NN \g__iow_streams_seq \g__ior_streams_seq
⟨/initex⟩
```

(*End definition for* \g__iow_streams_seq*. This variable is documented on page* **??**.)

\l__iow_stream_tl

Used to recover the raw stream number from the stack.

```
\tl_new:N \l__iow_stream_tl
```

(*End definition for* \l__iow_stream_tl*. This variable is documented on page* **??**.)

\g__iow_streams_prop

As for reads, but with more reserved as LaTeX 2_ε takes up a few here.

```
\prop_new:N \g__iow_streams_prop
⟨*package⟩
\prop_put:Nnn \g__iow_streams_prop { 0 } { LaTeX2e~reserved }
\prop_put:Nnn \g__iow_streams_prop { 1 } { LaTeX2e~reserved }
\prop_put:Nnn \g__iow_streams_prop { 2 } { LaTeX2e~reserved }
⟨/package⟩
```

(*End definition for* \g__iow_streams_prop*. This variable is documented on page* **??**.)

19.4 Stream management

\iow_new:N
\iow_new:c

Reserving a new stream is done by defining the name as equal to writing to the terminal: odd but at least consistent.

```
\cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
\cs_generate_variant:Nn \iow_new:N { c }
```

(*End definition for* \iow_new:N *and* \iow_new:c*. These functions are documented on page* **??**.)

\iow_open:Nn
\iow_open:cn
__iow_open:Nn
__iow_open_stream:Nn

The same idea as for reading, but without the path and without the need to allow for a conditional version.

```
\cs_new_protected:Npn \iow_open:Nn #1#2
  { \__file_name_sanitize:nn {#2} { \__iow_open:Nn #1 } }
\cs_generate_variant:Nn \iow_open:Nn { c }
\cs_new_protected:Npn \__iow_open:Nn #1#2
  {
    \iow_close:N #1
    \seq_gpop:NNTF \g__iow_streams_seq \l__iow_stream_tl
      { \__iow_open_stream:Nn #1 {#2} }
⟨*initex⟩
      { \__msg_kernel_fatal:nn { kernel } { output-streams-exhausted } }
⟨/initex⟩
⟨*package⟩
      {
        \cs:w newwrite \cs_end: #1
        \tl_set:Nx \l__iow_stream_tl { \int_eval:n {#1} }
        \__iow_open_stream:Nn #1 {#2}
      }
```

```
⟨/package⟩
  }
\cs_generate_variant:Nn \__iow_open:Nn { No }
\cs_new_protected:Npn \__iow_open_stream:Nn #1#2
  {
    \tex_global:D \tex_chardef:D #1 = \l__iow_stream_tl \scan_stop:
    \prop_gput:NVn \g__iow_streams_prop #1 {#2}
    \tex_immediate:D \tex_openout:D #1 #2 \scan_stop:
  }
```

(*End definition for* `\iow_open:Nn` *and* `\iow_open:cn`*. These functions are documented on page* **??**.)

`\iow_close:N`
`\iow_close:c`

Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.

```
\cs_new_protected:Npn \iow_close:N #1
  {
    \int_compare:nT { \c_minus_one < #1 < \c_sixteen }
      {
        \tex_immediate:D \tex_closeout:D #1
        \prop_gremove:NV \g__iow_streams_prop #1
        \seq_if_in:NVF \g__iow_streams_seq #1
          { \seq_gpush:NV \g__iow_streams_seq #1 }
        \cs_gset_eq:NN #1 \c_term_ior
      }
  }
\cs_generate_variant:Nn \iow_close:N { c }
```

(*End definition for* `\iow_close:N` *and* `\iow_close:c`*. These functions are documented on page* **??**.)

`\iow_list_streams:`
`\__iow_list_streams:Nn`

Done as for input, but with a copy of the auxiliary so the name is correct.

```
\cs_new_protected_nopar:Npn \iow_list_streams:
  { \__iow_list_streams:Nn \g__iow_streams_prop { output } }
\cs_new_eq:NN \__iow_list_streams:Nn \__ior_list_streams:Nn
```

(*End definition for* `\iow_list_streams:`*. This function is documented on page* **??**.)

19.4.1 Deferred writing

`\iow_shipout_x:Nn`
`\iow_shipout_x:Nx`

First the easy part, this is the primitive, which expects its argument to be braced.

```
\cs_new_protected:Npn \iow_shipout_x:Nn #1#2
  { \tex_write:D #1 {#2} }
\cs_generate_variant:Nn \iow_shipout_x:Nn { Nx }
```

(*End definition for* `\iow_shipout_x:Nn` *and* `\iow_shipout_x:Nx`*. These functions are documented on page* **??**.)

`\iow_shipout:Nn`
`\iow_shipout:Nx`

With ε-TeX available deferred writing without expansion is easy.

```
\cs_new_protected:Npn \iow_shipout:Nn #1#2
  { \tex_write:D #1 { \exp_not:n {#2} } }
\cs_generate_variant:Nn \iow_shipout:Nn { Nx }
```

(*End definition for* `\iow_shipout:Nn` *and* `\iow_shipout:Nx`*. These functions are documented on page* **??**.)

19.4.2 Immediate writing

\iow_now:Nn \iow_now:Nx 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.

```
\cs_new_protected:Npn \iow_now:Nn #1#2
  { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
\cs_generate_variant:Nn \iow_now:Nn { Nx }
```

(*End definition for* `\iow_now:Nn` *and* `\iow_now:Nx`*. These functions are documented on page* **??**.)

\iow_log:n \iow_log:x \iow_term:n \iow_term:x Writing to the log and the terminal directly are relatively easy.

```
\cs_set_protected_nopar:Npn \iow_log:x  { \iow_now:Nx \c_log_iow  }
\cs_new_protected_nopar:Npn \iow_log:n  { \iow_now:Nn \c_log_iow  }
\cs_set_protected_nopar:Npn \iow_term:x { \iow_now:Nx \c_term_iow }
\cs_new_protected_nopar:Npn \iow_term:n { \iow_now:Nn \c_term_iow }
```

(*End definition for* `\iow_log:n` *and* `\iow_log:x`*. These functions are documented on page* **??**.)

19.4.3 Special characters for writing

\iow_newline: Global variable holding the character that forces a new line when something is written to an output stream.

```
\cs_new_nopar:Npn \iow_newline: { ^^J }
```

(*End definition for* `\iow_newline:`*. This function is documented on page 166.*)

\iow_char:N Function to write any escaped char to an output stream.

```
\cs_new_eq:NN \iow_char:N \cs_to_str:N
```

(*End definition for* `\iow_char:N`*. This function is documented on page 166.*)

19.4.4 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.

```
\int_new:N  \l_iow_line_count_int
\int_set:Nn \l_iow_line_count_int { 78 }
```

(*End definition for* `\l_iow_line_count_int`*. This variable is documented on page 167.*)

\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.

```
\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.

```
\int_new:N \l__iow_current_line_int
\int_new:N \l__iow_current_word_int
\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.

```
\tl_new:N \l__iow_current_line_tl
\tl_new:N \l__iow_current_word_tl
\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.

```
\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*⟩.

```
\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.

```
\bool_new:N \l__iow_line_start_bool
```

(*End definition for* `\l__iow_line_start_bool`*.*)

\c_catcode_other_space_tl

Lowercase a character with category code 12 to produce an "other" space. We can do everything within the group, because `\tl_const:Nn` defines its argument globally.

```
\group_begin:
  \char_set_catcode_other:N \*
  \char_set_lccode:nn {`\*} {`\ }
  \tl_to_lowercase:n { \tl_const:Nn \c_catcode_other_space_tl { * } }
\group_end:
```

(*End definition for* `\c_catcode_other_space_tl`*. This function is documented on page* *168**.*)

\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.

```
\group_begin:
  \int_set_eq:NN \tex_escapechar:D \c_minus_one
  \tl_const:Nx \c__iow_wrap_marker_tl
    { \tl_to_str:n { \^^I \^^O \^^W \^^_ \^^W \^^R \^^A \^^P } }
\group_end:
\tl_map_inline:nn
```

```
    { { end } { newline } { indent } { unindent } }
    {
      \tl_const:cx { c__iow_wrap_ #1 _marker_tl }
        {
          \c_catcode_other_space_tl
          \c__iow_wrap_marker_tl
          \c_catcode_other_space_tl
          #1
          \c_catcode_other_space_tl
        }
    }
```

(*End definition for* `\c__iow_wrap_marker_tl`.)

`\iow_indent:n`
`\__iow_indent:n`

We give a dummy (protected) 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.

```
  \cs_new_protected:Npn \iow_indent:n #1 { }
  \cs_new:Npx \__iow_indent:n #1
    {
      \c__iow_wrap_indent_marker_tl
      #1
      \c__iow_wrap_unindent_marker_tl
    }
```

(*End definition for* `\iow_indent:n`*. This function is documented on page 167.*)

`\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 2_ε'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.

```
  \cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
    {
      \group_begin:
        \int_set_eq:NN \tex_escapechar:D \c_minus_one
        \cs_set_nopar:Npx \{ { \token_to_str:N \{ }
        \cs_set_nopar:Npx \# { \token_to_str:N \# }
        \cs_set_nopar:Npx \} { \token_to_str:N \} }
        \cs_set_nopar:Npx \% { \token_to_str:N \% }
        \cs_set_nopar:Npx \~ { \token_to_str:N \~ }
        \int_set:Nn \tex_escapechar:D { 92 }
        \cs_set_eq:NN \\ \c__iow_wrap_newline_marker_tl
```

```
      \cs_set_eq:NN \  \c_catcode_other_space_tl
      \cs_set_eq:NN \iow_indent:n \__iow_indent:n
      #3
⟨*initex⟩
      \tl_set:Nx \l__iow_wrap_tl {#1}
⟨/initex⟩
⟨*package⟩
      \__iow_wrap_set:Nx \l__iow_wrap_tl {#1}
⟨/package⟩
```

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.

```
      \tl_set:Nx \l__iow_newline_tl { \iow_newline: #2 }
      \tl_set:Nx \l__iow_newline_tl { \tl_to_str:N \l__iow_newline_tl }
      \tl_replace_all:Nnn \l__iow_newline_tl { ~ } { \c_space_tl }
      \int_set:Nn \l__iow_target_count_int
        { \l_iow_line_count_int - \tl_count:N \l__iow_newline_tl + \c_one }
      \int_zero:N \l__iow_current_indentation_int
      \tl_clear:N \l__iow_current_indentation_tl
      \int_zero:N \l__iow_current_line_int
      \tl_clear:N \l__iow_current_line_tl
      \bool_set_true:N \l__iow_line_start_bool
      \use:x
        {
          \exp_not:n { \tl_clear:N \l__iow_wrap_tl }
          \__iow_wrap_loop:w
          \tl_to_str:N \l__iow_wrap_tl
          \tl_to_str:N \c__iow_wrap_end_marker_tl
          \c_space_tl \c_space_tl
          \exp_not:N \q_stop
        }
    \exp_args:NNo \group_end:
    #4 \l__iow_wrap_tl
  }
```

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.

```
⟨*package⟩
\cs_new_eq:NN \__iow_wrap_set:Nx \protected@edef
⟨/package⟩
```

(*End definition for* `\iow_wrap:nnnN`*. This function is documented on page* *167**.*)

`\__iow_wrap_loop:w` The loop grabs one word in the input, and checks whether it is the special marker, or a normal word.

```
\cs_new_protected:Npn \__iow_wrap_loop:w #1 ~ %
  {
    \tl_set:Nn \l__iow_current_word_tl {#1}
    \tl_if_eq:NNTF \l__iow_current_word_tl \c__iow_wrap_marker_tl
```

```
      { \__iow_wrap_special:w }
      { \__iow_wrap_word: }
  }
```

(*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.

```
\cs_new_protected_nopar:Npn \__iow_wrap_word:
  {
    \int_set:Nn \l__iow_current_word_int
      { \__str_count_ignore_spaces:N \l__iow_current_word_tl }
    \int_add:Nn \l__iow_current_line_int { \l__iow_current_word_int }
    \int_compare:nNnTF \l__iow_current_line_int < \l__iow_target_count_int
      { \__iow_wrap_word_fits: }
      { \__iow_wrap_word_newline: }
    \__iow_wrap_loop:w
  }
\cs_new_protected_nopar:Npn \__iow_wrap_word_fits:
  {
    \bool_if:NTF \l__iow_line_start_bool
      {
        \bool_set_false:N \l__iow_line_start_bool
        \tl_put_right:Nx \l__iow_current_line_tl
          { \l__iow_current_indentation_tl \l__iow_current_word_tl }
        \int_add:Nn \l__iow_current_line_int
          { \l__iow_current_indentation_int }
      }
      {
        \tl_put_right:Nx \l__iow_current_line_tl
          { ~ \l__iow_current_word_tl }
        \int_incr:N \l__iow_current_line_int
      }
  }
\cs_new_protected_nopar:Npn \__iow_wrap_word_newline:
  {
    \tl_put_right:Nx \l__iow_wrap_tl
      { \l__iow_current_line_tl \l__iow_newline_tl }
    \int_set:Nn \l__iow_current_line_int
      {
        \l__iow_current_word_int
        + \l__iow_current_indentation_int
      }
    \tl_set:Nx \l__iow_current_line_tl
      { \l__iow_current_indentation_tl \l__iow_current_word_tl }
  }
```

(*End definition for* `\__iow_wrap_word:`.)

__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.

```
\cs_new_protected:Npn \__iow_wrap_special:w #1 ~ #2 ~ #3 ~ %
  {
    \use:c { __iow_wrap_#1: }
    \str_if_eq_x:nnTF { #2~#3 } { ~ \c__iow_wrap_marker_tl }
      { \__iow_wrap_special:w }
      { \__iow_wrap_loop:w #2 ~ #3 ~ }
  }
\cs_new_protected_nopar:Npn \__iow_wrap_newline:
  {
    \tl_put_right:Nx \l__iow_wrap_tl
      { \l__iow_current_line_tl \l__iow_newline_tl }
    \int_zero:N \l__iow_current_line_int
    \tl_clear:N \l__iow_current_line_tl
    \bool_set_true:N \l__iow_line_start_bool
  }
\cs_new_protected_nopar:Npx \__iow_wrap_indent:
  {
    \int_add:Nn \l__iow_current_indentation_int \c_four
    \tl_put_right:Nx \exp_not:N \l__iow_current_indentation_tl
      { \c_space_tl \c_space_tl \c_space_tl \c_space_tl }
  }
\cs_new_protected_nopar:Npn \__iow_wrap_unindent:
  {
    \int_sub:Nn \l__iow_current_indentation_int \c_four
    \tl_set:Nx \l__iow_current_indentation_tl
      { \prg_replicate:nn \l__iow_current_indentation_int { ~ } }
  }
\cs_new_protected_nopar:Npn \__iow_wrap_end:
  {
    \tl_put_right:Nx \l__iow_wrap_tl
      { \l__iow_current_line_tl }
    \use_none_delimit_by_q_stop:w
  }
```

(*End definition for* `\__iow_wrap_special:w`.)

__str_count_ignore_spaces:N
__str_count_ignore_spaces:n
__str_count_loop:NNNNNNNNN

The wrapping code requires to measure the number of character in each word. This could be done with `\tl_count:n`, but it is ten times faster (literally) to use the code below.

```
\cs_new_nopar:Npn \__str_count_ignore_spaces:N
  { \exp_args:No \__str_count_ignore_spaces:n }
\cs_new:Npn \__str_count_ignore_spaces:n #1
  {
    \__int_value:w \__int_eval:w
```

```
      \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1}
        { X8 } { X7 } { X6 } { X5 } { X4 } { X3 } { X2 } { X1 } { X0 } \q_stop
    \__int_eval_end:
  }
\cs_new:Npn \__str_count_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {
    \if_catcode:w X #9
      \exp_after:wN \use_none_delimit_by_q_stop:w
    \else:
      9 +
      \exp_after:wN \__str_count_loop:NNNNNNNNN
    \fi:
  }
```

(*End definition for* `\__str_count_ignore_spaces:N`.)

19.5 Messages

```
\__msg_kernel_new:nnnn { kernel } { file-not-found }
  { File~'#1'~not~found. }
  {
    The~requested~file~could~not~be~found~in~the~current~directory,~
    in~the~TeX~search~path~or~in~the~LaTeX~search~path.
  }
\__msg_kernel_new:nnnn { kernel } { input-streams-exhausted }
  { Input~streams~exhausted }
  {
    TeX~can~only~open~up~to~16~input~streams~at~one~time.\\
    All~16~are~currently~in~use,~and~something~wanted~to~open~
    another~one.
  }
\__msg_kernel_new:nnnn { kernel } { output-streams-exhausted }
  { Output~streams~exhausted }
  {
    TeX~can~only~open~up~to~16~output~streams~at~one~time.\\
    All~16~are~currently~in~use,~and~something~wanted~to~open~
    another~one.
  }
\__msg_kernel_new:nnnn { kernel } { space-in-file-name }
  { Space~in~file~name~'#1'. }
  {
    Spaces~are~not~permitted~in~files~loaded~by~LaTeX: \\
    Further~errors~may~follow!
  }
⟨/initex | package⟩
```

20 l3fp implementation

Nothing to see here: everything is in the subfiles!

21 l3fp-aux implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

22 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:[6]

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⟩`}` `;`

[6] 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 {$\langle exponent\rangle$} {$\langle X_1\rangle$} {$\langle X_2\rangle$} {$\langle X_3\rangle$} {$\langle X_4\rangle$} ;	Positive floating point.
1 2 {$\langle exponent\rangle$} {$\langle X_1\rangle$} {$\langle X_2\rangle$} {$\langle X_3\rangle$} {$\langle X_4\rangle$} ;	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`.

Here, the $\langle exponent\rangle$ is an integer, at most `\c__fp_max_exponent_int` = 10000 in absolute value. The body consists in four blocks of exactly 4 digits, $0000 \le \langle X_i\rangle \le 9999$, such that

$$\langle X\rangle = (-1)^{\langle sign\rangle} 10^{-\langle exponent\rangle} \sum_{i=1}^{4} \langle X_i\rangle 10^{-4i}$$

and such that the $\langle exponent\rangle$ is minimal. This implies $1000 \le \langle X_1\rangle \le 9999$.

23 Internal storage of floating points numbers

A floating point number $\langle X\rangle$ is stored as

`\s__fp \__fp_chk:w` $\langle case\rangle$ $\langle sign\rangle$ $\langle body\rangle$;

Here, $\langle case\rangle$ is 0 for ± 0, 1 for normal numbers, 2 for $\pm\infty$, and 3 for `nan`, and $\langle sign\rangle$ is 0 for positive numbers, 1 for `nan`s, and 2 for negative numbers. The $\langle body\rangle$ 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 $\langle exponent\rangle$ lies between $\pm$`\c__fp_max_exponent_int` = ± 10000 inclusive.

Additionally, positive and negative floating point numbers may only be stored with $1000 \le \langle X_1\rangle < 10000$. This requirement is necessary in order to preserve accuracy and speed.

23.1 Using arguments and semicolons

`\__fp_use_none_stop_f:n` This function removes an argument (typically a digit) and replaces it by `\exp_stop_f:`, a marker which stops `f`-type expansion.

```
\cs_new:Npn \__fp_use_none_stop_f:n #1 { \exp_stop_f: }
```

(*End definition for* `\__fp_use_none_stop_f:n`.)

__fp_use_s:n
__fp_use_s:nn

Those functions place a semicolon after one or two arguments (typically digits).

```
\cs_new:Npn \__fp_use_s:n #1 { #1; }
\cs_new:Npn \__fp_use_s:nn #1#2 { #1#2; }
```

(*End definition for* `\__fp_use_s:n` *and* `\__fp_use_s:nn`.)

__fp_use_none_until_s:w
__fp_use_i_until_s:nw
__fp_use_ii_until_s:nnw

Those functions select specific arguments among a set of arguments delimited by a semicolon.

```
\cs_new:Npn \__fp_use_none_until_s:w #1; { }
\cs_new:Npn \__fp_use_i_until_s:nw #1#2; {#1}
\cs_new:Npn \__fp_use_ii_until_s:nnw #1#2#3; {#2}
```

(*End definition for* `\__fp_use_none_until_s:w`*,* `\__fp_use_i_until_s:nw`*, and* `\__fp_use_ii_until_s:nnw`.)

__fp_reverse_args:Nww

Many internal functions take arguments delimited by semicolons, and it is occasionally useful to swap two such arguments.

```
\cs_new:Npn \__fp_reverse_args:Nww #1 #2; #3; { #1 #3; #2; }
```

(*End 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`.

```
\cs_new:Npn \__fp_rrot:www #1; #2; #3; { #2; #3; #1; }
```

(*End definition for* `\__fp_rrot:www`.)

__fp_use_i:ww
__fp_use_i:www

Many internal functions take arguments delimited by semicolons, and it is occasionally useful to remove one or two such arguments.

```
\cs_new:Npn \__fp_use_i:ww #1; #2; { #1; }
\cs_new:Npn \__fp_use_i:www #1; #2; #3; { #1; }
```

(*End definition for* `\__fp_use_i:ww` *and* `\__fp_use_i:www`.)

23.2 Constants, and structure of floating points

\s__fp
__fp_chk:w

Floating points numbers all start with `\s__fp \__fp_chk:w`, where `\s__fp` is equal to the TeX primitive `\relax`, and `\__fp_chk:w` is protected. The rest of the floating point number is made of characters (or `\relax`). This ensures that nothing expands under `f`-expansion, nor under `x`-expansion. However, when typeset, `\s__fp` does nothing, and `\__fp_chk:w` is expanded. We define `\__fp_chk:w` to produce an error.

```
\__scan_new:N \s__fp
\cs_new_protected:Npn \__fp_chk:w #1 ;
  {
    \__msg_kernel_error:nnx { kernel } { misused-fp }
      { \fp_to_tl:n { \s__fp \__fp_chk:w #1 ; } }
  }
```

(*End definition for* `\s__fp` *and* `\__fp_chk:w`.)

\s__fp_mark
\s__fp_stop

Aliases of `\tex_relax:D`, used to terminate expressions.

```
\__scan_new:N \s__fp_mark
\__scan_new:N \s__fp_stop
```

(*End definition for* `\s__fp_mark` *and* `\s__fp_stop`.)

\s__fp_invalid
\s__fp_underflow
\s__fp_overflow
\s__fp_division
\s__fp_exact

A couple of scan marks used to indicate where special floating point numbers come from.

```
\__scan_new:N \s__fp_invalid
\__scan_new:N \s__fp_underflow
\__scan_new:N \s__fp_overflow
\__scan_new:N \s__fp_division
\__scan_new:N \s__fp_exact
```

(*End definition for* `\s__fp_invalid` *and others.*)

\c_zero_fp
\c_minus_zero_fp
\c_inf_fp
\c_minus_inf_fp
\c_nan_fp

The special floating points. All of them have the form

`\s__fp \__fp_chk:w` ⟨*case*⟩ ⟨*sign*⟩ `\s__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".

```
\tl_const:Nn \c_zero_fp       { \s__fp \__fp_chk:w 0 0 \s__fp_exact ; }
\tl_const:Nn \c_minus_zero_fp { \s__fp \__fp_chk:w 0 2 \s__fp_exact ; }
\tl_const:Nn \c_inf_fp        { \s__fp \__fp_chk:w 2 0 \s__fp_exact ; }
\tl_const:Nn \c_minus_inf_fp  { \s__fp \__fp_chk:w 2 2 \s__fp_exact ; }
\tl_const:Nn \c_nan_fp        { \s__fp \__fp_chk:w 3 1 \s__fp_exact ; }
```

(*End definition for* `\c_zero_fp` *and others. These variables are documented on page* **??**.)

\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 $-$`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.

```
\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.

```
\cs_new:Npn \__fp_zero_fp:N #1 { \s__fp \__fp_chk:w 0 #1 \s__fp_underflow ; }
\cs_new:Npn \__fp_inf_fp:N #1  { \s__fp \__fp_chk:w 2 #1 \s__fp_overflow ; }
```

(*End 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.

```
\cs_new:Npn \__fp_min_fp:N #1
  {
    \s__fp \__fp_chk:w 1 #1
      { \int_eval:n { - \c__fp_max_exponent_int } }
      {1000} {0000} {0000} {0000} ;
  }
\cs_new:Npn \__fp_max_fp:N #1
  {
```

```
    \s__fp \__fp_chk:w 1 #1
      { \int_use:N \c__fp_max_exponent_int }
      {9999} {9999} {9999} {9999} ;
  }
```

(*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.

```
\cs_new:Npn \__fp_exponent:w \s__fp \__fp_chk:w #1
  {
    \if_meaning:w 1 #1
      \exp_after:wN \__fp_use_ii_until_s:nnw
    \else:
      \exp_after:wN \__fp_use_i_until_s:nw
      \exp_after:wN 0
    \fi:
  }
```

(*End definition for* `\__fp_exponent:w`.)

`\__fp_neg_sign:N` When appearing in an integer expression or after `\__int_value:w`, this expands to the sign opposite to `#1`, namely 0 (positive) is turned to 2 (negative), 1 (`nan`) to 1, and 2 to 0.

```
\cs_new:Npn \__fp_neg_sign:N #1
  { \__int_eval:w \c_two - #1 \__int_eval_end: }
```

(*End definition for* `\__fp_neg_sign:N`.)

23.3 Overflow, underflow, and exact zero

`\__fp_sanitize:Nw`
`\__fp_sanitize:wN`
`\__fp_sanitize_zero:w`

Expects the sign and the exponent in some order, then the significand (which we don't touch). Outputs the corresponding floating point number, possibly underflowed to ± 0 or overflowed to $\pm\infty$. The functions `\__fp_underflow:w` and `\__fp_overflow:w` are defined in l3fp-traps.

```
\cs_new:Npn \__fp_sanitize:Nw #1 #2;
  {
    \if_case:w \if_int_compare:w #2 > \c__fp_max_exponent_int \c_one \else:
               \if_int_compare:w #2 < - \c__fp_max_exponent_int \c_two \else:
               \if_meaning:w 1 #1 \c_three \else: \c_zero \fi: \fi: \fi:
    \or: \exp_after:wN \__fp_overflow:w
    \or: \exp_after:wN \__fp_underflow:w
    \or: \exp_after:wN \__fp_sanitize_zero:w
    \fi:
    \s__fp \__fp_chk:w 1 #1 {#2}
  }
\cs_new:Npn \__fp_sanitize:wN #1; #2 { \__fp_sanitize:Nw #2 #1; }
\cs_new:Npn \__fp_sanitize_zero:w \s__fp \__fp_chk:w #1 #2 #3; { \c_zero_fp }
```

(*End definition for* `\__fp_sanitize:Nw` *and* `\__fp_sanitize:wN`.)

23.4 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.

```
\cs_new:Npn \__fp_exp_after_o:w \s__fp \__fp_chk:w #1
  {
    \if_meaning:w 1 #1
      \exp_after:wN \__fp_exp_after_normal:nNNw
    \else:
      \exp_after:wN \__fp_exp_after_special:nNNw
    \fi:
    { }
    #1
  }
\cs_new:Npn \__fp_exp_after_o:nw #1 \s__fp \__fp_chk:w #2
  {
    \if_meaning:w 1 #2
      \exp_after:wN \__fp_exp_after_normal:nNNw
    \else:
      \exp_after:wN \__fp_exp_after_special:nNNw
    \fi:
    { #1 }
    #2
  }
\cs_new:Npn \__fp_exp_after_f:nw #1 \s__fp \__fp_chk:w #2
  {
    \if_meaning:w 1 #2
      \exp_after:wN \__fp_exp_after_normal:nNNw
    \else:
      \exp_after:wN \__fp_exp_after_special:nNNw
    \fi:
    { \tex_romannumeral:D -`0 #1 }
    #2
  }
```

(*End definition for* `\__fp_exp_after_o:w`.)

__fp_exp_after_special:nNNw

Special floating point numbers are easy to jump over since they contain few tokens.

```
\cs_new:Npn \__fp_exp_after_special:nNNw #1#2#3#4;
  {
    \exp_after:wN \s__fp
    \exp_after:wN \__fp_chk:w
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN #4
    \exp_after:wN ;
    #1
```

```
  }
```

(*End definition for* `\__fp_exp_after_special:nNNw`.)

`\__fp_exp_after_normal:nNNw` For normal floating point numbers, life is slightly harder, since we have many tokens to jump over. Here it would be slightly better if the digits were not braced but instead were delimited arguments (for instance delimited by `,`). That may be changed some day.

```
\cs_new:Npn \__fp_exp_after_normal:nNNw #1 1 #2 #3 #4#5#6#7;
  {
    \exp_after:wN \__fp_exp_after_normal:Nwwwww
    \exp_after:wN #2
    \__int_value:w #3   \exp_after:wN ;
    \__int_value:w 1 #4 \exp_after:wN ;
    \__int_value:w 1 #5 \exp_after:wN ;
    \__int_value:w 1 #6 \exp_after:wN ;
    \__int_value:w 1 #7 \exp_after:wN ; #1
  }
\cs_new:Npn \__fp_exp_after_normal:Nwwwww
    #1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;
  { \s__fp \__fp_chk:w 1 #1 {#2} {#3} {#4} {#5} {#6} ; }
```

(*End definition for* `\__fp_exp_after_normal:nNNw`.)

`\__fp_exp_after_array_f:w`
`\__fp_exp_after_stop_f:nw`

```
\cs_new:Npn \__fp_exp_after_array_f:w #1
  {
    \cs:w __fp_exp_after \__fp_type_from_scan:N #1 _f:nw \cs_end:
      { \__fp_exp_after_array_f:w }
    #1
  }
\cs_new_eq:NN \__fp_exp_after_stop_f:nw \use_none:nn
```

(*End definition for* `\__fp_exp_after_array_f:w`.)

23.5 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_use:N \__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_use:N \__int_eval:w - 5 0000
  \exp_after:wN \pack:NNNNNw
  \int_use:N \__int_eval:w 4 9995 0000
    + 12345 * 6677
    \exp_after:wN \pack:NNNNNw
    \int_use:N \__int_eval:w 5 0000 0000
      + 12345 * 8899 ;
```

The `\exp_after:wN` triggers `\int_use:N \__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_use:N \__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 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 $[-4\cdot10^8, 5\cdot10^8-1]$. Shifted values all have exactly 9 digits.

```
\int_const:Nn \c__fp_leading_shift_int  { - 5 0000 }
\int_const:Nn \c__fp_middle_shift_int   { 5 0000 *  9999 }
\int_const:Nn \c__fp_trailing_shift_int { 5 0000 * 10000 }
\cs_new:Npn \__fp_pack:NNNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }
```

(*End definition for* `\__fp_pack:NNNNNw`.)

`\__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 $[-5\cdot10^8, 6\cdot10^8-1]$ (actually a bit more). Shifted values all have exactly 10 digits. Note that the upper bound is due to TeX's limit of $2^{31}-1$ on integers. The shifts are chosen to be roughly the mid-point of 10^9 and 2^{31}, the two bounds on 10-digit integers in TeX.

```
\int_const:Nn \c__fp_big_leading_shift_int  { - 15 2374 }
\int_const:Nn \c__fp_big_middle_shift_int   { 15 2374 *  9999 }
\int_const:Nn \c__fp_big_trailing_shift_int { 15 2374 * 10000 }
\cs_new:Npn \__fp_pack_big:NNNNNNw #1#2 #3#4#5#6 #7;
  { + #1#2#3#4#5#6 ; {#7} }
```

(*End definition for* `\__fp_pack_big:NNNNNNw`.)

__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 involving results in the range $[-1\cdot 10^9, 147483647]$; the end-point is $2^{31} - 1 - 2\cdot 10^9 \simeq 1.47\cdot 10^8$. Shifted values all have exactly 10 digits.

```
\int_const:Nn \c__fp_Bigg_leading_shift_int  { - 20 0000 }
\int_const:Nn \c__fp_Bigg_middle_shift_int   { 20 0000 *  9999 }
\int_const:Nn \c__fp_Bigg_trailing_shift_int { 20 0000 * 10000 }
\cs_new:Npn \__fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7;
  { + #1#2#3#4#5#6 ; {#7} }
```

(*End definition for* `\__fp_pack_Bigg:NNNNNNw`*.*)

__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.

```
\cs_new:Npn \__fp_pack_twice_four:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
  { #1 {#2#3#4#5} {#6#7#8#9} ; }
```

(*End 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.

```
\cs_new:Npn \__fp_pack_eight:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
  { #1 {#2#3#4#5#6#7#8#9} ; }
```

(*End definition for* `\__fp_pack_eight:wNNNNNNNN`*.*)

23.6 Decimate (dividing by a power of 10)

__fp_decimate:nNnnnn

Each $\langle X_i\rangle$ consists in 4 digits exactly, and $1000 \le \langle X_1\rangle < 9999$. The first argument determines by how much we shift the digits. $\langle f_1\rangle$ is called as follows: where $0 \le \langle X'_i\rangle < 10^8 - 1$ 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 $\langle shift\rangle$.

```
\cs_new:Npn \__fp_decimate:nNnnnn #1
  {
    \cs:w
      __fp_decimate_
```

```
      \if_int_compare:w \__int_eval:w #1 > \c_sixteen
        tiny
      \else:
        \tex_romannumeral:D \__int_eval:w #1
      \fi:
      :Nnnnn
    \cs_end:
  }
```

Each of the auxiliaries see the function ⟨f_1⟩, followed by 4 blocks of 4 digits.
(*End definition for* `\__fp_decimate:nNnnnn`.)

`\__fp_decimate_:Nnnnn`
`\__fp_decimate_tiny:Nnnnn`

If the ⟨*shift*⟩ is zero, or too big, life is very easy.

```
\cs_new:Npn \__fp_decimate_:Nnnnn #1 #2#3#4#5
  { #1 0 {#2#3} {#4#5} ; }
\cs_new:Npn \__fp_decimate_tiny:Nnnnn #1 #2#3#4#5
  { #1 1 { 0000 0000 } { 0000 0000 } 0 #2#3#4#5 ; }
```

(*End 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 ⟨*rounding*⟩ digit, and repack digits into two blocks of 8. The sixteen functions are very similar, and defined through `\__fp_tmp:w`. The arguments are as follows: `#1` indicates which function is being defined; after one step of expansion, `#2` yields the "extra digits" which are then converted by `\__fp_round_digit:Nw` to the ⟨*rounding*⟩ digit. This triggers the `f`-expansion of `\__fp_decimate_pack:nnnnnnnnnnw`,[7] responsible for building two blocks of 8 digits, and removing the rest. For this to work, `#3` alternates between braced and unbraced blocks of 4 digits, in such a way that the 5 first and 5 next token groups yield the correct blocks of 8 digits.

```
\cs_new:Npn \__fp_tmp:w #1 #2 #3
  {
    \cs_new:cpn { __fp_decimate_ #1 :Nnnnn } ##1 ##2##3##4##5
      {
        \exp_after:wN ##1
        \__int_value:w
          \exp_after:wN \__fp_round_digit:Nw #2 ;
        \__fp_decimate_pack:nnnnnnnnnnw #3 ;
      }
  }
\__fp_tmp:w {i}   {\use_none:nnn      #50} {    0{#2}#3{#4}#5             }
\__fp_tmp:w {ii}  {\use_none:nn       #5 } {    00{#2}#3{#4}#5            }
\__fp_tmp:w {iii} {\use_none:n        #5 } {    000{#2}#3{#4}#5           }
\__fp_tmp:w {iv}  {                   #5 } {   {0000}#2{#3}#4 #5          }
\__fp_tmp:w {v}   {\use_none:nnn    #4#5 } {   0{0000}#2{#3}#4 #5         }
\__fp_tmp:w {vi}  {\use_none:nn     #4#5 } {   00{0000}#2{#3}#4 #5        }
\__fp_tmp:w {vii} {\use_none:n      #4#5 } {   000{0000}#2{#3}#4 #5       }
\__fp_tmp:w {viii}{                 #4#5 } {  {0000}0000{#2}#3 #4 #5      }
\__fp_tmp:w {ix}  {\use_none:nnn  #3#4+#5} {  0{0000}0000{#2}#3 #4 #5     }
\__fp_tmp:w {x}   {\use_none:nn   #3#4+#5} {  00{0000}0000{#2}#3 #4 #5    }
\__fp_tmp:w {xi}  {\use_none:n    #3#4+#5} {  000{0000}0000{#2}#3 #4 #5   }
```

[7] No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

```
\__fp_tmp:w {xii} {               #3#4+#5} { {0000}0000{0000}#2 #3 #4 #5     }
\__fp_tmp:w {xiii}{\use_none:nnn#2#3+#4#5} { 0{0000}0000{0000}#2 #3 #4 #5    }
\__fp_tmp:w {xiv} {\use_none:nn #2#3+#4#5} { 00{0000}0000{0000}#2 #3 #4 #5   }
\__fp_tmp:w {xv}  {\use_none:n  #2#3+#4#5} { 000{0000}0000{0000}#2 #3 #4 #5  }
\__fp_tmp:w {xvi} {             #2#3+#4#5} {{0000}0000{0000}0000 #2 #3 #4 #5 }
```

(*End 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.

```
\cs_new:Npn \__fp_decimate_pack:nnnnnnnnnnw #1#2#3#4#5
  { \__fp_decimate_pack:nnnnnnw { #1#2#3#4#5 } }
\cs_new:Npn \__fp_decimate_pack:nnnnnnw #1 #2#3#4#5#6
  { {#1} {#2#3#4#5#6} }
```

(*End definition for* `\__fp_round_digit:Nw` *and* `\__fp_decimate_pack:nnnnnnnnnnw`*.*)

23.7 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.

```
\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.

```
\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.

```
\cs_new:Npn \__fp_case_return_o:Nw #1#2 \fi: #3 \s__fp #4 ;
  { \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.

```
\cs_new:Npn \__fp_case_return_same_o:w #1 \fi: #2 \s__fp
  { \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.

```
\cs_new:Npn \__fp_case_return_o:Nww #1#2 \fi: #3 \s__fp #4 ; #5 ;
  { \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.

```
\cs_new:Npn \__fp_case_return_i_o:ww #1 \fi: #2 \s__fp #3 ; \s__fp #4 ;
  { \fi: \__fp_exp_after_o:w \s__fp #3 ; }
\cs_new:Npn \__fp_case_return_ii_o:ww #1 \fi: #2 \s__fp #3 ;
  { \fi: \__fp_exp_after_o:w }
```

(*End definition for* `\__fp_case_return_i_o:ww` *and* `\__fp_case_return_ii_o:ww`.)

23.8 Small integer floating points

`\__fp_small_int:wTF`
`\__fp_small_int_true:wTF`
`\__fp_small_int_normal:NnwTF`
`\__fp_small_int_test:NnnwNTF`

Tests if the floating point argument is an integer or $\pm\infty$. If so, it is 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.

```
\cs_new:Npn \__fp_small_int:wTF \s__fp \__fp_chk:w #1#2
  {
    \if_case:w #1 \exp_stop_f:
           \__fp_case_return:nw { \__fp_small_int_true:wTF 0 ; }
    \or:   \exp_after:wN \__fp_small_int_normal:NnwTF
    \or:
      \__fp_case_return:nw
        {
          \exp_after:wN \__fp_small_int_true:wTF \__int_value:w
            \if_meaning:w 2 #2 - \fi: 1 0000 0000 ;
        }
    \else: \__fp_case_return:nw \use_ii:nn
    \fi:
    #2
  }
\cs_new:Npn \__fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
\cs_new:Npn \__fp_small_int_normal:NnwTF #1#2#3;
  {
    \if_int_compare:w #2 > \c_zero
      \__fp_decimate:nNnnnn { \c_sixteen - #2 }
        \__fp_small_int_test:NnnwNnw
        #3 #1 {#2}
    \else:
      \exp_after:wN \use_iii:nnn
    \fi:
    ;
  }
\cs_new:Npn \__fp_small_int_test:NnnwNnw #1#2#3#4; #5#6
  {
    \if_meaning:w 0 #1
      \exp_after:wN \__fp_small_int_true:wTF
      \__int_value:w \if_meaning:w 2 #5 - \fi:
        \if_int_compare:w #6 > \c_eight
          1 0000 0000
        \else:
          #3
        \fi:
    \else:
      \use_i:nn
    \fi:
  }
```

(*End definition for* `\__fp_small_int:wTF`.)

23.9 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.

```
\cs_new:Npn \__fp_array_count:n #1
  {
    \int_use:N \__int_eval:w \c_zero
      \__fp_array_count_loop:Nw #1 { ? \__prg_break: } ;
      \__prg_break_point:
    \__int_eval_end:
  }
\cs_new:Npn \__fp_array_count_loop:Nw #1#2;
  { \use_none:n #1 + \c_one \__fp_array_count_loop:Nw }
```

(*End definition for* __fp_array_count:n.)

23.10 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.

```
\cs_new:Npn \__fp_expand:n #1
  {
    \__fp_expand_loop:nwnN { }
      #1 \prg_do_nothing:
      \s__fp_mark { } \__fp_expand_loop:nwnN
      \s__fp_mark { } \__fp_use_i_until_s:nw ;
  }
\cs_new:Npn \__fp_expand_loop:nwnN #1#2 \s__fp_mark #3 #4
  {
    \exp_after:wN #4 \tex_romannumeral:D -`0
    #2
    \s__fp_mark { #3 #1 } #4
  }
```

(*End definition for* __fp_expand:n.)

23.11 Messages

Using a floating point directly is an error.

```
\__msg_kernel_new:nnnn { kernel } { misused-fp }
  { A~floating~point~with~value~'#1'~was~misused. }
  {
    To~obtain~the~value~of~a~floating~point~variable,~use~
    '\token_to_str:N \fp_to_decimal:N',~
    '\token_to_str:N \fp_to_scientific:N',~or~other~
    conversion~functions.
  }
⟨/initex | package⟩
```

24 l3fp-traps Implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

Exceptions should be accessed by an `n`-type argument, among

- `invalid_operation`
- `division_by_zero`
- `overflow`
- `underflow`
- `inexact` (actually never used).

24.1 Flags

`\fp_flag_off:n` Function to turn a flag off. Simply undefine it.

```
\cs_new_protected:Npn \fp_flag_off:n #1
  { \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 177.*)

`\fp_flag_on:n` Function to turn a flag on expandably: use TeX's automatic assignment to `\scan_stop:`.

```
\cs_new:Npn \fp_flag_on:n #1
  { \exp_args:Nc \use_none:n { l__fp_ #1 _flag_token } }
```

(*End definition for* `\fp_flag_on:n`. *This function is documented on page 177.*)

`\fp_if_flag_on_p:n`
`\fp_if_flag_on:nTF`

Returns true if the flag is on, false otherwise.

```
\prg_new_conditional:Npnn \fp_if_flag_on:n #1 { p , T , F , TF }
  {
    \if_cs_exist:w l__fp_ #1 _flag_token \cs_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\fp_if_flag_on:n`. *These functions are documented on page 177.*)

`\l__fp_invalid_operation_flag_token`
`\l__fp_division_by_zero_flag_token`
`\l__fp_overflow_flag_token`
`\l__fp_underflow_flag_token`

The IEEE standard defines five exceptions. We currently don't support the "inexact" exception.

```
\cs_new_eq:NN \l__fp_invalid_operation_flag_token \tex_undefined:D
\cs_new_eq:NN \l__fp_division_by_zero_flag_token \tex_undefined:D
\cs_new_eq:NN \l__fp_overflow_flag_token \tex_undefined:D
\cs_new_eq:NN \l__fp_underflow_flag_token \tex_undefined:D
```

(*End definition for* `\l__fp_invalid_operation_flag_token` *and others.*)

24.2 Traps

Exceptions can be trapped to obtain custom behaviour. When an invalid operation or a division by zero is trapped, the trap receives as arguments the result as an N-type floating point number, the function name (multiple letters for prefix operations, or a single symbol for infix operations), and the operand(s). When an overflow or underflow is trapped, the trap receives the resulting overly large or small floating point number if it is not too big, otherwise it receives $+\infty$. Currently, the inexact exception is entirely ignored.

The behaviour when an exception occurs is controlled by the definitions of the functions

- `\__fp_invalid_operation:nnw`,
- `\__fp_invalid_operation_o:Nww`,
- `\__fp_invalid_operation_tl_o:ff`,
- `\__fp_division_by_zero_o:Nnw`,
- `\__fp_division_by_zero_o:NNww`,
- `\__fp_overflow:w`,
- `\__fp_underflow:w`.

Rather than changing them directly, we provide a user interface as `\fp_trap:nn {`⟨*exception*⟩`} {`⟨*way of trapping*⟩`}`, where the ⟨*way of trapping*⟩ is one of `error`, `flag`, or `none`.

We also provide `\__fp_invalid_operation_o:nw`, defined in terms of `\__fp_invalid_operation:nnw`.

\fp_trap:nn

```
\cs_new_protected:Npn \fp_trap:nn #1#2
  {
    \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
      {
        \clist_if_in:nnTF
          { invalid_operation , division_by_zero , overflow , underflow }
          {#1}
          {
            \__msg_kernel_error:nnxx { kernel }
              { unknown-fpu-trap-type } {#1} {#2}
          }
          { \__msg_kernel_error:nnx { kernel } { unknown-fpu-exception } {#1} }
      }
  }
```

(*End definition for* `\fp_trap:nn`. *This function is documented on page 178.*)

__fp_trap_invalid_operation_set_error:
__fp_trap_invalid_operation_set_flag:
__fp_trap_invalid_operation_set_none:
__fp_trap_invalid_operation_set:N

We provide three types of trapping for invalid operations: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In most cases, the function produces as a result its first argument, possibly with post-expansion.

```
\cs_new_protected_nopar:Npn \__fp_trap_invalid_operation_set_error:
  { \__fp_trap_invalid_operation_set:N \prg_do_nothing: }
\cs_new_protected_nopar:Npn \__fp_trap_invalid_operation_set_flag:
  { \__fp_trap_invalid_operation_set:N \use_none:nnnnn }
\cs_new_protected_nopar:Npn \__fp_trap_invalid_operation_set_none:
  { \__fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
\cs_new_protected:Npn \__fp_trap_invalid_operation_set:N #1
  {
    \exp_args:Nno \use:n
      { \cs_set:Npn \__fp_invalid_operation:nnw ##1##2##3; }
      {
        #1
        \__fp_error:nnfn { invalid } {##2} { \fp_to_tl:n { ##3; } } { }
        \fp_flag_on:n { invalid_operation }
        ##1
      }
    \exp_args:Nno \use:n
      { \cs_set:Npn \__fp_invalid_operation_o:Nww ##1##2; ##3; }
      {
        #1
        \__fp_error:nffn { invalid-ii }
          { \fp_to_tl:n { ##2; } } { \fp_to_tl:n { ##3; } } {##1}
        \fp_flag_on:n { invalid_operation }
        \exp_after:wN \c_nan_fp
      }
    \exp_args:Nno \use:n
      { \cs_set:Npn \__fp_invalid_operation_tl_o:ff ##1##2 }
      {
        #1
        \__fp_error:nffn { invalid } {##1} {##2} { }
        \fp_flag_on:n { invalid_operation }
        \exp_after:wN \c_nan_fp
      }
  }
```

(*End definition for* `\__fp_trap_invalid_operation_set_error:` *and others.*)

__fp_trap_division_by_zero_set_error:
__fp_trap_division_by_zero_set_flag:
__fp_trap_division_by_zero_set_none:
__fp_trap_division_by_zero_set:N

We provide three types of trapping for invalid operations and division by zero: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In all cases, the function must produce a result, namely its first argument, $\pm\infty$ or `nan`.

```
\cs_new_protected_nopar:Npn \__fp_trap_division_by_zero_set_error:
  { \__fp_trap_division_by_zero_set:N \prg_do_nothing: }
\cs_new_protected_nopar:Npn \__fp_trap_division_by_zero_set_flag:
  { \__fp_trap_division_by_zero_set:N \use_none:nnnnn }
\cs_new_protected_nopar:Npn \__fp_trap_division_by_zero_set_none:
  { \__fp_trap_division_by_zero_set:N \use_none:nnnnnnn }
```

```
\cs_new_protected:Npn \__fp_trap_division_by_zero_set:N #1
  {
    \exp_args:Nno \use:n
      { \cs_set:Npn \__fp_division_by_zero_o:Nnw ##1##2##3; }
      {
        #1
        \__fp_error:nnfn { zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
        \fp_flag_on:n { division_by_zero }
        \exp_after:wN ##1
      }
    \exp_args:Nno \use:n
      { \cs_set:Npn \__fp_division_by_zero_o:NNww ##1##2##3; ##4; }
      {
        #1
        \__fp_error:nffn { zero-div-ii }
          { \fp_to_tl:n { ##3; } } { \fp_to_tl:n { ##4; } } {##2}
        \fp_flag_on:n { division_by_zero }
        \exp_after:wN ##1
      }
  }
```

(*End definition for* `\__fp_trap_division_by_zero_set_error:` *and others.*)

__fp_trap_overflow_set_error:
__fp_trap_overflow_set_flag:
__fp_trap_overflow_set_none:
__fp_trap_overflow_set:N
__fp_trap_underflow_set_error:
__fp_trap_underflow_set_flag:
__fp_trap_underflow_set_none:
__fp_trap_underflow_set:N
__fp_trap_overflow_set:NnNn

Just as for invalid operations and division by zero, the three different behaviours are obtained by feeding `\prg_do_nothing:`, `\use_none:nnnnn` or `\use_none:nnnnnnn` to an auxiliary, with a further auxiliary common to overflow and underflow functions. In most cases, the argument of the `\__fp_overflow:w` and `\__fp_underflow:w` functions will be an (almost) normal number (with an exponent outside the allowed range), and the error message thus displays that number together with the result to which it overflowed or underflowed. For extreme cases such as `10 ** 1e9999`, the exponent would be too large for TeX, and `\__fp_overflow:w` receives $\pm\infty$ (`\__fp_underflow:w` would receive ± 0); then we cannot do better than simply say an overflow or underflow occurred.

```
\cs_new_protected_nopar:Npn \__fp_trap_overflow_set_error:
  { \__fp_trap_overflow_set:N \prg_do_nothing: }
\cs_new_protected_nopar:Npn \__fp_trap_overflow_set_flag:
  { \__fp_trap_overflow_set:N \use_none:nnnnn }
\cs_new_protected_nopar:Npn \__fp_trap_overflow_set_none:
  { \__fp_trap_overflow_set:N \use_none:nnnnnnn }
\cs_new_protected:Npn \__fp_trap_overflow_set:N #1
  { \__fp_trap_overflow_set:NnNn #1 { overflow } \__fp_inf_fp:N { inf } }
\cs_new_protected_nopar:Npn \__fp_trap_underflow_set_error:
  { \__fp_trap_underflow_set:N \prg_do_nothing: }
\cs_new_protected_nopar:Npn \__fp_trap_underflow_set_flag:
  { \__fp_trap_underflow_set:N \use_none:nnnnn }
\cs_new_protected_nopar:Npn \__fp_trap_underflow_set_none:
  { \__fp_trap_underflow_set:N \use_none:nnnnnnn }
\cs_new_protected:Npn \__fp_trap_underflow_set:N #1
  { \__fp_trap_overflow_set:NnNn #1 { underflow } \__fp_zero_fp:N { 0 } }
\cs_new_protected:Npn \__fp_trap_overflow_set:NnNn #1#2#3#4
  {
```

```
    \exp_args:Nno \use:n
      { \cs_set:cpn { __fp_ #2 :w } \s__fp \__fp_chk:w ##1##2##3; }
      {
        #1
        \__fp_error:nffn
          { flow \if_meaning:w 1 ##1 -to \fi: }
          { \fp_to_tl:n { \s__fp \__fp_chk:w ##1##2##3; } }
          { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }
          {#2}
        \fp_flag_on:n {#2}
        #3 ##2
      }
  }
```

(*End definition for* `\__fp_trap_overflow_set_error:` *and others.*)

`\__fp_invalid_operation:nnw`
`\__fp_invalid_operation_o:Nww`
`\__fp_invalid_operation_tl_o:ff`
`\__fp_division_by_zero_o:Nnw`
`\__fp_division_by_zero_o:NNww`
`\__fp_overflow:w`
`\__fp_underflow:w`

Initialize the two control sequences (to log properly their existence). Then set invalid operations to trigger an error, and division by zero, overflow, and underflow to act silently on their flag.

```
\cs_new:Npn \__fp_invalid_operation:nnw #1#2#3; { }
\cs_new:Npn \__fp_invalid_operation_o:Nww #1#2; #3; { }
\cs_new:Npn \__fp_invalid_operation_tl_o:ff #1 #2 { }
\cs_new:Npn \__fp_division_by_zero_o:Nnw #1#2#3; { }
\cs_new:Npn \__fp_division_by_zero_o:NNww #1#2#3; #4; { }
\cs_new:Npn \__fp_overflow:w { }
\cs_new:Npn \__fp_underflow:w { }
\fp_trap:nn { invalid_operation } { error }
\fp_trap:nn { division_by_zero } { flag }
\fp_trap:nn { overflow } { flag }
\fp_trap:nn { underflow } { flag }
```

(*End definition for* `\__fp_invalid_operation:nnw` *and others.*)

`\__fp_invalid_operation_o:nw`
`\__fp_invalid_operation_o:fw`

Convenient short-hands for returning `\c_nan_fp` for a unary or binary operation, and expanding after.

```
\cs_new_nopar:Npn \__fp_invalid_operation_o:nw
  { \__fp_invalid_operation:nnw { \exp_after:wN \c_nan_fp } }
\cs_generate_variant:Nn \__fp_invalid_operation_o:nw { f }
```

(*End definition for* `\__fp_invalid_operation_o:nw` *and* `\__fp_invalid_operation_o:fw`*.*)

24.3 Errors

`\__fp_error:nnnn`
`\__fp_error:nnfn`
`\__fp_error:nffn`

```
\cs_new:Npn \__fp_error:nnnn #1
  { \__msg_kernel_expandable_error:nnnnn { kernel } { fp - #1 } }
\cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff }
```

(*End definition for* `\__fp_error:nnnn`*,* `\__fp_error:nnfn`*, and* `\__fp_error:nffn`*.*)

24.4 Messages

Some messages.

```
\__msg_kernel_new:nnnn { kernel } { unknown-fpu-exception }
  { The~FPU~exception~'#1'~is~not~known:~that~trap~will~never~be~triggered. }
  {
    The~only~exceptions~to~which~traps~can~be~attached~are \\
    \iow_indent:n
      {
        * ~ invalid_operation \\
        * ~ division_by_zero \\
        * ~ overflow \\
        * ~ underflow
      }
  }
\__msg_kernel_new:nnnn { kernel } { unknown-fpu-trap-type }
  { The~FPU~trap~type~'#2'~is~not~known. }
  {
    The~trap~type~must~be~one~of \\
    \iow_indent:n
      {
        * ~ error \\
        * ~ flag \\
        * ~ none
      }
  }
\__msg_kernel_new:nnn { kernel } { fp-flow }
  { An ~ #3 ~ occurred. }
\__msg_kernel_new:nnn { kernel } { fp-flow-to }
  { #1 ~ #3 ed ~ to ~ #2 . }
\__msg_kernel_new:nnn { kernel } { fp-zero-div }
  { Division~by~zero~in~ #1 (#2) }
\__msg_kernel_new:nnn { kernel } { fp-zero-div-ii }
  { Division~by~zero~in~ (#1) #3 (#2) }
\__msg_kernel_new:nnn { kernel } { fp-invalid }
  { Invalid~operation~ #1 (#2) }
\__msg_kernel_new:nnn { kernel } { fp-invalid-ii }
  { Invalid~operation~ (#1) #3 (#2) }
```

⟨/initex | package⟩

25 l3fp-round implementation

⟨*initex | package⟩

⟨@@=fp⟩

25.1 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*⟩ ⟨*digit*$_1$⟩ ⟨*digit*$_2$⟩ can expand to `\c_zero` or `\c_one`.
- `\__fp_round_s:NNNw` ⟨*sign*⟩ ⟨*digit*$_1$⟩ ⟨*digit*$_2$⟩ ⟨*more digits*⟩ `;` can expand to `\c_zero ;` or `\c_one ;`.
- `\__fp_round_neg:NNN` ⟨*sign*⟩ ⟨*digit*$_1$⟩ ⟨*digit*$_2$⟩ can expand to `\c_zero` or `\c_one`.

See implementation comments for details on the syntax.

`\__fp_round:NNN`
`\__fp_round_to_nearest:NNN`
`\__fp_round_to_ninf:NNN`
`\__fp_round_to_zero:NNN`
`\__fp_round_to_pinf:NNN`

If rounding the number ⟨*final sign*⟩⟨*digit*$_1$⟩.⟨*digit*$_2$⟩ 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 ⟨*final sign*⟩ 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 ⟨*final sign*⟩ 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 ⟨*digit*$_2$⟩ is larger than 5, then round up. If it is less than 5, round down. If it is exactly 5, then round such that ⟨*digit*$_1$⟩ plus the result is even. In other words, round up if ⟨*digit*$_1$⟩ is odd.

```
\cs_new:Npn \__fp_round_return_one:
```

```
  { \exp_after:wN \c_one \tex_romannumeral:D }
\cs_new:Npn \__fp_round_to_ninf:NNN #1 #2 #3
  {
    \if_meaning:w 2 #1
      \if_int_compare:w #3 > \c_zero
        \__fp_round_return_one:
      \fi:
    \fi:
    \c_zero
  }
\cs_new:Npn \__fp_round_to_zero:NNN #1 #2 #3 { \c_zero }
\cs_new:Npn \__fp_round_to_pinf:NNN #1 #2 #3
  {
    \if_meaning:w 0 #1
      \if_int_compare:w #3 > \c_zero
        \__fp_round_return_one:
      \fi:
    \fi:
    \c_zero
  }
\cs_new:Npn \__fp_round_to_nearest:NNN #1 #2 #3
  {
    \if_int_compare:w #3 > \c_five
      \__fp_round_return_one:
    \else:
      \if_meaning:w 5 #3
        \if_int_odd:w #2 \exp_stop_f:
          \__fp_round_return_one:
        \fi:
      \fi:
    \fi:
    \c_zero
  }
\cs_new_eq:NN \__fp_round:NNN \__fp_round_to_nearest:NNN
```

(*End definition for* `\__fp_round:NNN`.)

`\__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.

```
\cs_new:Npn \__fp_round_s:NNNw #1 #2 #3 #4;
  {
    \exp_after:wN \__fp_round:NNN
    \exp_after:wN #1
    \exp_after:wN #2
    \int_use:N \__int_eval:w
      \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
                      \if_meaning:w 5 #3 1 \fi:
```

```
                \exp_stop_f:
        \if_int_compare:w \__int_eval:w #4 > \c_zero
          1 +
        \fi:
      \fi:
      #3
    ;
  }
```

(*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.

```
\cs_new:Npn \__fp_round_digit:Nw #1 #2;
  {
    \if_int_odd:w \if_meaning:w 0 #1 \c_one \else:
                  \if_meaning:w 5 #1 \c_one \else:
                  \c_zero \fi: \fi:
      \if_int_compare:w \__int_eval:w #2 > \c_zero
        \__int_eval:w \c_one +
      \fi:
    \fi:
    #1
  }
```

(*End definition for* `\__fp_round_digit:Nw`.)

`\__fp_round_neg:NNN`
`\__fp_round_to_nearest_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`. Consider a number of the form ⟨*final sign*⟩.$X \ldots X$⟨$digit_1$⟩ with exactly 15 (non-all-zero) digits before ⟨$digit_1$⟩, and subtract from it ⟨*final sign*⟩.$0 \ldots 0$⟨$digit_2$⟩, 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.

```
\cs_new:Npn \__fp_round_to_ninf_neg:NNN #1 #2 #3
  {
    \if_meaning:w 0 #1
      \if_int_compare:w #3 > \c_zero
        \__fp_round_return_one:
      \fi:
    \fi:
    \c_zero
  }
\cs_new:Npn \__fp_round_to_zero_neg:NNN #1 #2 #3
  {
    \if_int_compare:w #3 > \c_zero
      \__fp_round_return_one:
    \fi:
```

```
    \c_zero
  }
\cs_new:Npn \__fp_round_to_pinf_neg:NNN #1 #2 #3
  {
    \if_meaning:w 2 #1
      \if_int_compare:w #3 > \c_zero
        \__fp_round_return_one:
      \fi:
    \fi:
    \c_zero
  }
\cs_new_eq:NN \__fp_round_to_nearest_neg:NNN \__fp_round_to_nearest:NNN
\cs_new_eq:NN \__fp_round_neg:NNN \__fp_round_to_nearest_neg:NNN
```

(*End definition for* `\__fp_round_neg:NNN`.)

25.2 The round function

`\__fp_round_o:Nw`

This function expects one or two arguments.

```
\cs_new:Npn \__fp_round_o:Nw #1#2 @
  {
    \if_case:w
      \__int_eval:w \__fp_array_count:n {#2} - \c_one \__int_eval_end:
         \__fp_round:Nwn #1 #2 {0} \tex_romannumeral:D
    \or: \__fp_round:Nww #1 #2 \tex_romannumeral:D
    \else:
      \__fp_error:nffn { num-args }
        { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
      \exp_after:wN \c_nan_fp \tex_romannumeral:D
    \fi:
    -`0
  }
```

(*End definition for* `\__fp_round_o:Nw`.)

`\__fp_round_name_from_cs:N`

```
\cs_new:Npn \__fp_round_name_from_cs:N #1
  {
    \cs_if_eq:NNTF #1 \__fp_round_to_zero:NNN { trunc }
      {
        \cs_if_eq:NNTF #1 \__fp_round_to_ninf:NNN { floor }
          {
            \cs_if_eq:NNTF #1 \__fp_round_to_pinf:NNN { ceil }
              { round }
          }
      }
  }
```

(*End 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`

```
\cs_new:Npn \__fp_round:Nww #1#2 ; #3 ;
```

```
  {
    \__fp_small_int:wTF #3; { \__fp_round:Nwn #1#2; }
      {
        \__fp_invalid_operation_tl_o:ff
          { \__fp_round_name_from_cs:N #1 }
          { \__fp_array_to_clist:n { #2; #3; } }
      }
  }
\cs_new:Npn \__fp_round:Nwn #1 \s__fp \__fp_chk:w #2#3#4; #5
  {
    \if_meaning:w 1 #2
      \exp_after:wN \__fp_round_normal:NwNNnw
      \exp_after:wN #1
      \__int_value:w #5
    \else:
      \exp_after:wN \__fp_exp_after_o:w
    \fi:
    \s__fp \__fp_chk:w #2#3#4;
  }
\cs_new:Npn \__fp_round_normal:NwNNnw #1#2 \s__fp \__fp_chk:w 1#3#4#5;
  {
    \__fp_decimate:nNnnnn { \c_sixteen - #4 - #2 }
      \__fp_round_normal:NnnwNNnn #5 #1 #3 {#4} {#2}
  }
\cs_new:Npn \__fp_round_normal:NnnwNNnn #1#2#3#4; #5#6
  {
    \exp_after:wN \__fp_round_normal:NNwNnn
    \int_use:N \__int_eval:w
      \if_int_compare:w #2 > \c_zero
        1 \__int_value:w #2
        \exp_after:wN \__fp_round_pack:Nw
        \int_use:N \__int_eval:w 1#3 +
      \else:
        \if_int_compare:w #3 > \c_zero
          1 \__int_value:w #3 +
        \fi:
      \fi:
      \exp_after:wN #5
      \exp_after:wN #6
      \use_none:nnnnnnn #3
      #1
      \__int_eval_end:
      0000 0000 0000 0000 ; #6
  }
\cs_new:Npn \__fp_round_pack:Nw #1
  { \if_meaning:w 2 #1 + \c_one \fi: \__int_eval_end: }
\cs_new:Npn \__fp_round_normal:NNwNnn #1 #2
  {
    \if_meaning:w 0 #2
      \exp_after:wN \__fp_round_special:NwwNnn
```

```
      \exp_after:wN #1
    \fi:
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_round_normal_end:wwNnn
    ; #2
  }
\cs_new:Npn \__fp_round_normal_end:wwNnn #1;#2;#3#4#5
  {
    \exp_after:wN \__fp_exp_after_o:w \tex_romannumeral:D -`0
    \__fp_sanitize:Nw #3 #4 ; #1 ;
  }
\cs_new:Npn \__fp_round_special:NwwNnn #1#2;#3;#4#5#6
  {
    \if_meaning:w 0 #1
      \__fp_case_return:nw
        { \exp_after:wN \__fp_zero_fp:N \exp_after:wN #4 }
    \else:
      \exp_after:wN \__fp_round_special_aux:Nw
      \exp_after:wN #4
      \int_use:N \__int_eval:w \c_one
        \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
    \fi:
    ;
  }
\cs_new:Npn \__fp_round_special_aux:Nw #1#2;
  {
    \exp_after:wN \__fp_exp_after_o:w \tex_romannumeral:D -`0
    \__fp_sanitize:Nw #1#2; {1000}{0000}{0000}{0000};
  }
```

(*End definition for* `\__fp_round:Nww` *and* `\__fp_round:Nwn`.)

```
⟨/initex | package⟩
```

26 l3fp-parse implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

26.1 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.

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.

32 Juxtaposition for implicit multiplication.

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 %.

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.

26.1.1 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 ;
\tex_romannumeral:D \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 `\tex_romannumeral:D`, 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 ;
\tex_romannumeral:D \c_zero 333444 ;
```

where in fact `\exp_after:wN` has already been expanded, `\__int_value:w` has already seen `12345`, and `\tex_romannumeral:D` 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.

26.1.2 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⟩ ⟨number₂⟩
@ \__fp_parse_infix_⟨operator₂⟩:N
```

The `infix` function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand ⟨*number*⟩ is before the `infix` function in the input stream. This is why we stop the expansion here and give control to another function to close the loop.

A definition of `\__fp_parse_operand:Nw` ⟨*precedence*⟩ with some of the expansion control removed is

```
\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN ⟨precedence⟩
\tex_romannumeral:D -‘0
\__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*⟩
> `\tex_romannumeral:D -‘0`
> `\__fp_`⟨*operator*⟩`_o:ww` ⟨*number*⟩ ⟨*number*$_2$⟩
> `\tex_romannumeral:D -‘0`
> `\__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.

26.1.3 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 $3^{-2\times4}$ 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, except for juxtaposition.

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.

26.1.4 Numbers and reading tokens one by one

So far, we have glossed over one important point: what is a "number"? A number is typically given in the form ⟨*significand*⟩`e`⟨*exponent*⟩, where the ⟨*significand*⟩ is any non-empty string composed of decimal digits and at most one decimal separator (a period), the exponent "`e`⟨*exponent*⟩" is optional and is composed of an exponent mark `e` followed by a possibly empty string of signs `+` or `-` and a non-empty string of decimal digits. The ⟨*significand*⟩ can also be an integer, dimension, skip, or muskip variable, in which case dimensions are converted from points (or mu units) to floating points, and the ⟨*exponent*⟩

can also be an integer variable. Numbers can also be given as floating point variables, or as named constants such as `nan`, `inf` or `pi`. We may add more types in the future.

When `\__fp_parse_one:Nw` is looking for a "number", here is what happens.

- If the next token is a control sequence with the meaning of `\scan_stop:`, it can be: `\s__fp`, in which case our job is done, as what follows is an internal floating point number, or `\s__fp_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 end-marker `\s__fp_mark`, which has the lowest precedence, 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.

26.2 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 ⟨*operation*$_2$⟩ 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.)

26.3 Helpers

__fp_parse_expand:w

This function must always come within a `\romannumeral` expansion. The ⟨*tokens*⟩ should be the part of the expression that we have not yet read. This requires in particular closing all conditionals properly before expanding.

```
\cs_new:Npn \__fp_parse_expand:w #1 { -‘0 #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.

```
\cs_new:Npn \__fp_parse_return_semicolon:w
    #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 `_?`.

```
\group_begin:
\char_set_catcode_other:N \S
\char_set_catcode_other:N \F
\char_set_catcode_other:N \P
\char_set_lccode:nn { ‘\- } { ‘\_ }
\tl_to_lowercase:n
  {
    \group_end:
    \cs_new:Npn \__fp_type_from_scan:N #1
      {
        \exp_after:wN \__fp_type_from_scan:w
        \token_to_str:N #1 \q_mark S--FP-? \q_mark \q_stop
      }
    \cs_new:Npn \__fp_type_from_scan:w #1 S--FP #2 \q_mark #3 \q_stop {#2}
  }
```

(*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

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.

```
\cs_set_protected:Npn \__fp_tmp:w #1 #2 #3
  {
    \cs_new:cpn { __fp_parse_digits_ #1 :N } ##1
      {
        \if_int_compare:w \c_nine < 1 \token_to_str:N ##1 \exp_stop_f:
          \token_to_str:N ##1 \exp_after:wN #2 \tex_romannumeral:D
        \else:
          \__fp_parse_return_semicolon:w #3 ##1
        \fi:
        \__fp_parse_expand:w
      }
  }
\__fp_tmp:w {vii}  \__fp_parse_digits_vi:N   { 0000000 ; 7 }
\__fp_tmp:w {vi}   \__fp_parse_digits_v:N    { 000000 ; 6 }
\__fp_tmp:w {v}    \__fp_parse_digits_iv:N   { 00000 ; 5 }
\__fp_tmp:w {iv}   \__fp_parse_digits_iii:N  { 0000 ; 4 }
\__fp_tmp:w {iii}  \__fp_parse_digits_ii:N   { 000 ; 3 }
\__fp_tmp:w {ii}   \__fp_parse_digits_i:N    { 00 ; 2 }
\__fp_tmp:w {i}    \__fp_parse_digits_:N     { 0 ; 1 }
\cs_new_nopar:Npn \__fp_parse_digits_:N { ; ; 0 }
```

(*End definition for* `\__fp_parse_digits_vii:N` *and others.*)

26.4 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 happens with the LaTeX 2_ε command `\protect`. Testing if `#2` is a control sequence thus includes `\exp_not:N`.

```
\cs_new:Npn \__fp_parse_one:Nw #1 #2
  {
    \if_catcode:w \scan_stop: \exp_not:N #2
      \if_meaning:w \scan_stop: #2
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_one_fp:NN
      \else:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_one_register:NN
      \fi:
```

```
    \else:
      \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_one_digit:NN
      \else:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_parse_one_other:NN
      \fi:
    \fi:
    #1 #2
  }
```

(*End 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.

```
\cs_new:Npn \__fp_parse_one_fp:NN #1#2
  {
    \cs:w __fp_exp_after \__fp_type_from_scan:N #2 _f:nw \cs_end:
      {
        \exp_after:wN \__fp_parse_infix:NN
        \exp_after:wN #1 \tex_romannumeral:D \__fp_parse_expand:w
      }
    #2
  }
\cs_new:Npn \__fp_exp_after_mark_f:nw #1
  {
    \__msg_kernel_expandable_error:nn { kernel } { fp-early-end }
    \exp_after:wN \c_nan_fp \tex_romannumeral:D -`0 #1
  }
\cs_new:cpn { __fp_exp_after_?_f:nw } #1#2
  {
    \__msg_kernel_expandable_error:nnn { kernel } { bad-variable } {#2}
    \exp_after:wN \c_nan_fp \tex_romannumeral:D -`0 #1
  }
```

(*End definition for* `\__fp_parse_one_fp:NN`*,* `\__fp_exp_after_mark_f:nw`*, and* `\__fp_exp_after_?_f:nw`*.*)

__fp_parse_one_register:NN
__fp_parse_one_register_aux:Nw
__fp_parse_one_register_auxii:wwwNw
__fp_parse_one_register_int:www
__fp_parse_one_register_mu:www
__fp_parse_one_register_dim:ww

This is called whenever `#2` is a control sequence other than `\scan_stop:` in meaning. We 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.

```
\cs_new:Npn \__fp_parse_one_register:NN #1#2
  {
    \exp_after:wN \__fp_parse_infix_after_operand:NwN
    \exp_after:wN #1
    \tex_romannumeral:D -`0
      \exp_after:wN \__fp_parse_one_register_aux:Nw
      \exp_after:wN #2
      \__int_value:w
        \exp_after:wN \__fp_parse_exponent:N
        \tex_romannumeral:D \__fp_parse_expand:w
  }
\group_begin:
\char_set_catcode_other:N \P
\char_set_catcode_other:N \T
\char_set_catcode_other:N \M
\char_set_catcode_other:N \U
\tl_to_lowercase:n
  {
    \group_end:
    \cs_new:Npn \__fp_parse_one_register_aux:Nw #1
      {
        \exp_after:wN \use:nn
        \exp_after:wN \__fp_parse_one_register_auxii:wwwNw
        \exp_after:wN { \tex_the:D \exp_not:N #1 }
          ; \__fp_parse_one_register_dim:ww
          PT ; \__fp_parse_one_register_mu:www
          . PT ; \__fp_parse_one_register_int:www
          \q_stop
      }
    \cs_new:Npn \__fp_parse_one_register_auxii:wwwNw
        #1 . #2 PT #3 ; #4#5 \q_stop { #4 #1.#2; }
    \cs_new:Npn \__fp_parse_one_register_mu:www #1 MU; #2;
      { \__fp_parse_one_register_dim:ww #1; }
  }
\cs_new:Npn \__fp_parse_one_register_int:www #1; #2.; #3;
  { \__fp_parse:n { #1 e #3 } }
\cs_new:Npn \__fp_parse_one_register_dim:ww #1; #2;
  {
```

```
    \exp_after:wN \__fp_from_dim_test:ww
    \__int_value:w #2 \exp_after:wN ,
    \__int_value:w \__dim_eval:w #1 pt ;
  }
```

(*End definition for* `\__fp_parse_one_register:NN` *and others.*)

`\__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.

```
\cs_new:Npn \__fp_parse_one_digit:NN #1
  {
    \exp_after:wN \__fp_parse_infix_after_operand:NwN
    \exp_after:wN #1
    \tex_romannumeral:D -`0
      \exp_after:wN \__fp_sanitize:wN
      \int_use:N \__int_eval:w \c_zero \__fp_parse_trim_zeros:N
  }
```

(*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_`⟨*operator*⟩`:Nw`.

```
\cs_new:Npn \__fp_parse_one_other:NN #1 #2
  {
    \if_int_compare:w
        \__int_eval:w
          ( `#2 \if_int_compare:w `#2 > `Z - \c_thirty_two \fi: ) / 26
        = \c_three
      \exp_after:wN \__fp_parse_word:Nw
      \exp_after:wN #1
      \exp_after:wN #2
      \tex_romannumeral:D \exp_after:wN \__fp_parse_letters:N
      \tex_romannumeral:D
    \else:
      \exp_after:wN \__fp_parse_prefix:NNN
      \exp_after:wN #1
      \exp_after:wN #2
      \cs:w __fp_parse_prefix_#2:Nw \exp_after:wN \cs_end:
      \tex_romannumeral:D
    \fi:
    \__fp_parse_expand:w
  }
```

(*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.

```
\cs_new:Npn \__fp_parse_word:Nw #1#2;
  {
    \cs_if_exist_use:cF { __fp_parse_word_#2:N }
      {
        \__msg_kernel_expandable_error:nnn
          { kernel } { unknown-fp-word } {#2}
        \exp_after:wN \c_nan_fp \tex_romannumeral:D -‘0
        \__fp_parse_infix:NN
      }
      #1
  }
\cs_new:Npn \__fp_parse_letters:N #1
  {
    -‘0
    \if_int_compare:w
        \if_catcode:w \scan_stop: \exp_not:N #1
          \c_zero
        \else:
          \__int_eval:w
            ( ‘#1 \if_int_compare:w ‘#1 > ‘Z - \c_thirty_two \fi: )
            / 26
        \fi:
        = \c_three
      \exp_after:wN #1
      \tex_romannumeral:D \exp_after:wN \__fp_parse_letters:N
      \tex_romannumeral:D
    \else:
      \__fp_parse_return_semicolon:w #1
    \fi:
    \__fp_parse_expand:w
  }
```

(*End definition for __fp_parse_word:Nw.*)

__fp_parse_prefix:NNN
__fp_parse_prefix_unknown:NNN

For this function, #1 is the previous ⟨*precedence*⟩, #2 is the operator just seen, and #3 is a control sequence which implements the operator if it is a known operator. If this control sequence is \scan_stop:, then the operator is in fact unknown. Either the expression is missing a number there (if the operator is valid as an infix operator), and we put nan, wrapping the infix operator in a csname as appropriate, or the character is simply invalid in floating point expressions, and we continue looking for a number, starting again from __fp_parse_one:Nw.

```
\cs_new:Npn \__fp_parse_prefix:NNN #1#2#3
  {
```

```
    \if_meaning:w \scan_stop: #3
      \exp_after:wN \__fp_parse_prefix_unknown:NNN
      \exp_after:wN #2
    \fi:
    #3 #1
  }
\cs_new:Npn \__fp_parse_prefix_unknown:NNN #1#2#3
  {
    \cs_if_exist:cTF { __fp_parse_infix_#1:N }
      {
        \__msg_kernel_expandable_error:nnn
          { kernel } { fp-missing-number } {#1}
        \exp_after:wN \c_nan_fp \tex_romannumeral:D -`0
        \__fp_parse_infix:NN #3 #1
      }
      {
        \__msg_kernel_expandable_error:nnn
          { kernel } { fp-unknown-symbol } {#1}
        \__fp_parse_one:Nw #3
      }
  }
```

(*End definition for* `\__fp_parse_prefix:NNN` *and* `\__fp_parse_prefix_unknown:NNN`.)

26.4.1 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.

```
\cs_new:Npn \__fp_parse_trim_zeros:N #1
  {
    \if:w 0 \exp_not:N #1
      \exp_after:wN \__fp_parse_trim_zeros:N
      \tex_romannumeral:D
    \else:
      \if:w . \exp_not:N #1
        \exp_after:wN \__fp_parse_strim_zeros:N
        \tex_romannumeral:D
      \else:
        \__fp_parse_trim_end:w #1
      \fi:
```

```
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_trim_end:w #1 \fi: \fi: \__fp_parse_expand:w
  {
      \fi:
    \fi:
    \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
      \exp_after:wN \__fp_parse_large:N
    \else:
      \exp_after:wN \__fp_parse_zero:
    \fi:
    #1
  }
```

(*End definition for* `\__fp_parse_trim_zeros:N` *and* `\__fp_parse_trim_end:w`.)

`\__fp_parse_strim_zeros:N`
`\__fp_parse_strim_end:w`

If we have removed all digits until a period (or if the body started with a period), then enter the "`small_trim`" loop which outputs -1 for each removed 0. Those -1 are added to an integer expression waiting for the exponent. If the first non-zero token is a digit, call `\__fp_parse_small:N` (our significand is smaller than 1), and otherwise, the number is an exact zero. The name `strim` stands for "small trim".

```
\cs_new:Npn \__fp_parse_strim_zeros:N #1
  {
    \if:w 0 \exp_not:N #1
      - \c_one
      \exp_after:wN \__fp_parse_strim_zeros:N \tex_romannumeral:D
    \else:
      \__fp_parse_strim_end:w #1
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_strim_end:w #1 \fi: \__fp_parse_expand:w
  {
    \fi:
    \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
      \exp_after:wN \__fp_parse_small:N
    \else:
      \exp_after:wN \__fp_parse_zero:
    \fi:
    #1
  }
```

(*End 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).

```
\cs_new:Npn \__fp_parse_zero:
  {
    \exp_after:wN ; \exp_after:wN 1
    \__int_value:w \__fp_parse_exponent:N
```

```
  }
```

(*End definition for* `\__fp_parse_zero:`.)

26.4.2 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. Then the `pack_leading` auxiliary puts the various parts in the appropriate order for the processing further up.

```
\cs_new:Npn \__fp_parse_small:N #1
  {
    \exp_after:wN \__fp_parse_pack_leading:NNNNNww
    \int_use:N \__int_eval:w 1 \token_to_str:N #1
      \exp_after:wN \__fp_parse_small_leading:wwNN
      \__int_value:w 1
        \exp_after:wN \__fp_parse_digits_vii:N
        \tex_romannumeral:D \__fp_parse_expand:w
  }
```

(*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.

```
\cs_new:Npn \__fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
  {
    #1 #2
    \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
    \exp_after:wN \c_zero
    \int_use:N \__int_eval:w 1
      \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
        \token_to_str:N #4
        \exp_after:wN \__fp_parse_small_trailing:wwNN
        \__int_value:w 1
          \exp_after:wN \__fp_parse_digits_vi:N
          \tex_romannumeral:D
      \else:
        0000 0000 \__fp_parse_exponent:Nw #4
      \fi:
      \__fp_parse_expand:w
  }
```

(*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.

```
\cs_new:Npn \__fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
  {
    #1 #2
    \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
      \token_to_str:N #4
      \exp_after:wN \__fp_parse_small_round:NN
      \exp_after:wN #4
      \tex_romannumeral:D
    \else:
      0 \__fp_parse_exponent:Nw #4
    \fi:
    \__fp_parse_expand:w
  }
```

(*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.

```
\cs_new:Npn \__fp_parse_pack_trailing:NNNNNNww #1 #2 #3#4#5#6 #7; #8 ;
  {
    \if_meaning:w 2 #2 + \c_one \fi:
    ; #8 + #1 ; {#3#4#5#6} {#7};
  }
\cs_new:Npn \__fp_parse_pack_leading:NNNNNww #1 #2#3#4#5 #6; #7;
  {
    + #7
    \if_meaning:w 2 #1 \__fp_parse_pack_carry:w \fi:
    ; 0 {#2#3#4#5} {#6}
  }
\cs_new:Npn \__fp_parse_pack_carry:w \fi: ; 0 #1
  { \fi: + \c_one ; 0 {1000} }
```

(*End definition for* `\__fp_parse_pack_trailing:NNNNNNww`, `\__fp_parse_pack_leading:NNNNNww`, *and* `\__fp_parse_pack_carry:w`.)

26.4.3 Number: large significand

Parsing a significand larger than 1 is a little bit more difficult than parsing small significands. We need to count the number of digits before the decimal separator, and add that to the final exponent. We also need to test for the presence of a dot each time we run out of digits, and branch to the appropriate `parse_small` function in those cases.

__fp_parse_large:N This function is followed by the first non-zero digit of a "large" significand (≥ 1). It is called within an integer expression for the exponent. Grab up to 7 more digits, for a total of 8 digits.

```
\cs_new:Npn \__fp_parse_large:N #1
  {
    \exp_after:wN \__fp_parse_large_leading:wwNN
    \__int_value:w 1 \token_to_str:N #1
      \exp_after:wN \__fp_parse_digits_vii:N
      \tex_romannumeral:D \__fp_parse_expand:w
  }
```

(*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.

```
\cs_new:Npn \__fp_parse_large_leading:wwNN 1 #1 ; #2; #3 #4
  {
    + \c_eight - #3
    \exp_after:wN \__fp_parse_pack_leading:NNNNNww
    \int_use:N \__int_eval:w 1 #1
      \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
        \exp_after:wN \__fp_parse_large_trailing:wwNN
        \__int_value:w 1 \token_to_str:N #4
          \exp_after:wN \__fp_parse_digits_vi:N
          \tex_romannumeral:D
      \else:
        \if:w . \exp_not:N #4
          \exp_after:wN \__fp_parse_small_leading:wwNN
          \__int_value:w 1
            \cs:w
              __fp_parse_digits_
              \tex_romannumeral:D #3
              :N \exp_after:wN
            \cs_end:
            \tex_romannumeral:D
        \else:
          #2
          \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
          \exp_after:wN \c_zero
```

```
          \__int_value:w 1 0000 0000
          \__fp_parse_exponent:Nw #4
        \fi:
      \fi:
      \__fp_parse_expand:w
  }
```

(*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.

```
\cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
  {
    \if_int_compare:w \c_nine < 1 \token_to_str:N #4 \exp_stop_f:
      \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
      \exp_after:wN \c_eight
      \int_use:N \__int_eval:w 1 #1 \token_to_str:N #4
        \exp_after:wN \__fp_parse_large_round:NN
        \exp_after:wN #4
        \tex_romannumeral:D
    \else:
      \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
      \int_use:N \__int_eval:w \c_seven - #3 \exp_stop_f:
      \int_use:N \__int_eval:w 1 #1
        \if:w . \exp_not:N #4
          \exp_after:wN \__fp_parse_small_trailing:wwNN
          \__int_value:w 1
            \cs:w
              __fp_parse_digits_
              \tex_romannumeral:D #3
              :N \exp_after:wN
            \cs_end:
            \tex_romannumeral:D
        \else:
          #2 0 \__fp_parse_exponent:Nw #4
        \fi:
    \fi:
    \__fp_parse_expand:w
  }
```

(*End definition for* `\__fp_parse_large_trailing:wwNN`.)

26.4.4 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.

```
\cs_new:Npn \__fp_parse_round_loop:N #1
  {
    \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
      + \c_one
      \if:w 0 \token_to_str:N #1
        \exp_after:wN \__fp_parse_round_loop:N
        \tex_romannumeral:D
      \else:
        \exp_after:wN \__fp_parse_round_up:N
        \tex_romannumeral:D
      \fi:
    \else:
      \__fp_parse_return_semicolon:w \c_zero #1
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_round_up:N #1
  {
    \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
      + \c_one
      \exp_after:wN \__fp_parse_round_up:N
      \tex_romannumeral:D
    \else:
      \__fp_parse_return_semicolon:w \c_one #1
    \fi:
    \__fp_parse_expand:w
  }
```

(*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.

```
\cs_new:Npn \__fp_parse_round_after:wN #1; #2
  {
    + #2 \exp_after:wN ;
    \int_use:N \__int_eval:w #1 + \__fp_parse_exponent:N
  }
```

(*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.

```
\cs_new:Npn \__fp_parse_small_round:NN #1#2
  {
    \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
      +
      \exp_after:wN \__fp_round_s:NNNw
      \exp_after:wN 0
      \exp_after:wN #1
      \exp_after:wN #2
      \int_use:N \__int_eval:w
        \exp_after:wN \__fp_parse_round_after:wN
        \int_use:N \__int_eval:w \c_zero * \__int_eval:w \c_zero
          \exp_after:wN \__fp_parse_round_loop:N
          \tex_romannumeral:D
    \else:
      \__fp_parse_exponent:Nw #2
    \fi:
    \__fp_parse_expand:w
  }
```

(*End definition for __fp_parse_small_round:NN and __fp_parse_round_after:wN.*)

__fp_parse_large_round:NN
__fp_parse_large_round_test:NN
__fp_parse_large_round_aux:wNN

Large numbers are harder to round, as there may be a period in the way. Again, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If there are no more digits (#2 is not a digit), then we must test for a period: if there is one, then switch to the rounding function for small significands, otherwise fetch an exponent. If there are more digits (#2 is a digit), then round, checking with __fp_parse_round_loop:N if all further digits vanish, or some are non-zero. This loop is not enough, as it is stopped by a period. After the loop, the aux function tests for a period: if it is present, then we must continue looking for digits, this time discarding the number of digits we find.

```
\cs_new:Npn \__fp_parse_large_round:NN #1#2
  {
    \if_int_compare:w \c_nine < 1 \token_to_str:N #2 \exp_stop_f:
      +
      \exp_after:wN \__fp_round_s:NNNw
      \exp_after:wN 0
      \exp_after:wN #1
      \exp_after:wN #2
      \int_use:N \__int_eval:w
        \exp_after:wN \__fp_parse_large_round_aux:wNN
```

```
        \int_use:N \__int_eval:w \c_one
          \exp_after:wN \__fp_parse_round_loop:N
    \else: %^^A could be dot, or e, or other
      \exp_after:wN \__fp_parse_large_round_test:NN
      \exp_after:wN #1
      \exp_after:wN #2
    \fi:
  }
\cs_new:Npn \__fp_parse_large_round_test:NN #1#2
  {
    \if:w . \exp_not:N #2
      \exp_after:wN \__fp_parse_small_round:NN
      \exp_after:wN #1
      \tex_romannumeral:D
    \else:
      \__fp_parse_exponent:Nw #2
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_large_round_aux:wNN #1 ; #2 #3
  {
    + #2
    \exp_after:wN \__fp_parse_round_after:wN
    \int_use:N \__int_eval:w #1
      \if:w . \exp_not:N #3
        + \c_zero * \__int_eval:w \c_zero
          \exp_after:wN \__fp_parse_round_loop:N
          \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
      \else:
        \exp_after:wN ;
        \exp_after:wN \c_zero
        \exp_after:wN #3
      \fi:
  }
```

(*End definition for* `\__fp_parse_large_round:NN`*,* `\__fp_parse_large_round_test:NN`*, and* `\__fp_parse_large_round_aux:w`

26.4.5 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.

```
\cs_new:Npn \__fp_parse_exponent:Nw #1 #2 \__fp_parse_expand:w
  {
    \exp_after:wN ;
    \__int_value:w #2 \__fp_parse_exponent:N #1
  }
```

(*End definition for* `\__fp_parse_exponent:Nw`.)

`\__fp_parse_exponent:N`
`\__fp_parse_exponent_aux:N`

This function should be called within an `\__int_value:w` expansion (or within an integer expression. It leaves digits of the exponent behind it in the input stream, and terminates the expansion with a semicolon. If there is no `e`, 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.

```
\cs_new:Npn \__fp_parse_exponent:N #1
  {
    \if:w e \exp_not:N #1
      \exp_after:wN \__fp_parse_exponent_aux:N
      \tex_romannumeral:D
    \else:
      0 \__fp_parse_return_semicolon:w #1
    \fi:
    \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_exponent_aux:N #1
  {
    \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #1
                 \c_zero \else: ‘#1 \fi: > ‘9 \exp_stop_f:
      0 \exp_after:wN ; \exp_after:wN e
    \else:
      \exp_after:wN \__fp_parse_exponent_sign:N
    \fi:
    #1
  }
```

(*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).

```
\cs_new:Npn \__fp_parse_exponent_sign:N #1
  {
    \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
      \exp_after:wN \__fp_parse_exponent_sign:N
      \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
    \else:
      \exp_after:wN \__fp_parse_exponent_body:N
      \exp_after:wN #1
    \fi:
  }
```

(*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).

```
\cs_new:Npn \__fp_parse_exponent_body:N #1
  {
    \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
      \token_to_str:N #1
      \exp_after:wN \__fp_parse_exponent_digits:N
      \tex_romannumeral:D
    \else:
      \__fp_parse_exponent_keep:NTF #1
        { \__fp_parse_return_semicolon:w #1 }
        {
          \exp_after:wN ;
          \tex_romannumeral:D
        }
    \fi:
    \__fp_parse_expand:w
  }
```

(*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.

```
\cs_new:Npn \__fp_parse_exponent_digits:N #1
  {
    \if_int_compare:w \c_nine < 1 \token_to_str:N #1 \exp_stop_f:
      \token_to_str:N #1
      \exp_after:wN \__fp_parse_exponent_digits:N
      \tex_romannumeral:D
    \else:
      \__fp_parse_return_semicolon:w #1
    \fi:
    \__fp_parse_expand:w
```

```
  }
```

(*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.

```
\prg_new_conditional:Npnn \__fp_parse_exponent_keep:N #1 { TF }
  {
    \if_catcode:w \scan_stop: \exp_not:N #1
      \if_meaning:w \scan_stop: #1
        \if_int_compare:w
            \pdftex_strcmp:D { \s__fp } { \exp_not:N #1 } = \c_zero
          0
          \__msg_kernel_expandable_error:nnn
            { kernel } { fp-after-e } { floating~point~ }
          \prg_return_true:
        \else:
          0
          \__msg_kernel_expandable_error:nnn
            { kernel } { bad-variable } {#1}
          \prg_return_false:
        \fi:
      \else:
        \if_int_compare:w
            \pdftex_strcmp:D { \__int_value:w #1 } { \tex_the:D #1 }
            = \c_zero
          \__int_value:w #1
        \else:
          0
          \__msg_kernel_expandable_error:nnn
            { kernel } { fp-after-e } { dimension~#1 }
        \fi:
        \prg_return_false:
      \fi:
    \else:
      0
      \__msg_kernel_expandable_error:nnn
        { kernel } { fp-missing } { exponent }
      \prg_return_true:
    \fi:
  }
```

(*End definition for* `\__fp_parse_exponent_keep:NTF`.)

26.5 Constants, functions and prefix operators

26.5.1 Prefix operators

__fp_parse_prefix_+:Nw A unary + does nothing: we should continue looking for a number.

```
\cs_new_eq:cN { __fp_parse_prefix_+:Nw } \__fp_parse_one:Nw
```

(*End 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 \tex_romannumeral:D thus the infix function #5.

```
\cs_new:Npn \__fp_parse_apply_unary:NNNwN #1#2#3#4@#5
  {
    #3 #2 #4 @
    \tex_romannumeral:D -`0 #5 #1
  }
```

(*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.

```
\cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
  {
    \cs_new:cpn { __fp_parse_prefix_ #1 :Nw } ##1
      {
        \exp_after:wN \__fp_parse_apply_unary:NNNwN
        \exp_after:wN ##1
        \exp_after:wN #4
        \exp_after:wN #3
        \tex_romannumeral:D
        \if_int_compare:w #2 < ##1
          \__fp_parse_operand:Nw ##1
        \else:
          \__fp_parse_operand:Nw #2
        \fi:
        \__fp_parse_expand:w
      }
  }
\__fp_tmp:w - \c_twelve \__fp_set_sign_o:w 2
\__fp_tmp:w ! \c_twelve \__fp_not_o:w ?
```

(*End definition for* __fp_parse_prefix_-:Nw *and* __fp_parse_prefix_!:Nw.)

__fp_parse_prefix_.:Nw Numbers which start with a decimal separator (a period) end up here. Of course, we do not look for an operand, but for the rest of the number. This function is very similar to __fp_parse_one_digit:NN but calls __fp_parse_strim_zeros:N to trim zeros after the decimal point, rather than the `trim_zeros` function for zeros before the decimal point.

```
\cs_new:cpn { __fp_parse_prefix_.:Nw } #1
  {
    \exp_after:wN \__fp_parse_infix_after_operand:NwN
    \exp_after:wN #1
    \tex_romannumeral:D -`0
      \exp_after:wN \__fp_sanitize:wN
      \int_use:N \__int_eval:w \c_zero \__fp_parse_strim_zeros:N
  }
```

(*End definition for* `\__fp_parse_prefix_.:Nw`.)

`\__fp_parse_prefix_(:Nw`
`\__fp_parse_lparen_after:NwN`

The left parenthesis is treated as a unary prefix operator because it appears in exactly the same settings. Commas will be allowed if the previous precedence is 16 (function with multiple arguments) or 13 (unary boolean "not"). 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.

```
\group_begin:
  \char_set_catcode_letter:N (
  \char_set_catcode_letter:N )
  \cs_new:Npn \__fp_parse_prefix_(:Nw #1
    {
      \exp_after:wN \__fp_parse_lparen_after:NwN
      \exp_after:wN #1
      \tex_romannumeral:D
      \if_int_compare:w #1 = \c_sixteen
        \__fp_parse_operand:Nw \c_one
      \else:
        \__fp_parse_operand:Nw \c_zero
      \fi:
      \__fp_parse_expand:w
    }
  \cs_new:Npn \__fp_parse_lparen_after:NwN #1#2 @ #3
    {
      \token_if_eq_meaning:NNTF #3 \__fp_parse_infix_):N
        {
          \__fp_exp_after_array_f:w #2 \s__fp_stop
          \exp_after:wN \__fp_parse_infix:NN
          \exp_after:wN #1
          \tex_romannumeral:D \__fp_parse_expand:w
        }
        {
          \__msg_kernel_expandable_error:nnn
            { kernel } { fp-missing } { ) }
          #2 @ \use_none:n #3
        }
    }
```

```
\group_end:
```

(*End definition for* `\__fp_parse_prefix_(:Nw` *and* `\__fp_parse_lparen_after:NwN`.)

26.5.2 Constants

__fp_parse_word_inf:N
__fp_parse_word_nan:N
__fp_parse_word_pi:N
__fp_parse_word_deg:N
__fp_parse_word_true:N
__fp_parse_word_false:N

Some words correspond to constant floating points. The floating point constant is left as a result of `\__fp_parse_one:Nw` after expanding `\__fp_parse_infix:NN`.

```
\cs_set_protected:Npn \__fp_tmp:w #1 #2
  {
    \cs_new_nopar:cpn { __fp_parse_word_#1:N }
      { \exp_after:wN #2 \tex_romannumeral:D -`0 \__fp_parse_infix:NN }
  }
\__fp_tmp:w { inf } \c_inf_fp
\__fp_tmp:w { nan } \c_nan_fp
\__fp_tmp:w { pi  } \c_pi_fp
\__fp_tmp:w { deg } \c_one_degree_fp
\__fp_tmp:w { true } \c_one_fp
\__fp_tmp:w { false } \c_zero_fp
```

(*End 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.

```
\cs_set_protected:Npn \__fp_tmp:w #1 #2
  {
    \cs_new_nopar:cpn { __fp_parse_word_#1:N }
      {
        \__fp_exp_after_f:nw { \__fp_parse_infix:NN }
        \s__fp \__fp_chk:w 10 #2 ;
      }
  }
\__fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
\__fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
\__fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
\__fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
\__fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
\__fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
\__fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
\__fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
\__fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
\__fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
\__fp_tmp:w {sp} { {-4} {1525} {8789} {0625} {0000} }
```

(*End definition for* `\__fp_parse_word_pt:N` *and others.*)

__fp_parse_word_em:N
__fp_parse_word_ex:N

The font-dependent units `em` and `ex` must be evaluated on the fly. We reuse an auxiliary of `\dim_to_fp:n`.

```
\tl_map_inline:nn { {em} {ex} }
  {
    \cs_new_nopar:cpn { __fp_parse_word_#1:N }
```

```
      {
        \exp_after:wN \__fp_from_dim_test:ww
        \exp_after:wN 0 \exp_after:wN ,
        \__int_value:w \__dim_eval:w 1 #1 \exp_after:wN ;
        \tex_romannumeral:D -`0 \__fp_parse_infix:NN
      }
  }
```

(*End definition for* `\__fp_parse_word_em:N` *and* `\__fp_parse_word_ex:N`.)

26.5.3 Functions

`\__fp_parse_unary_function:nNN`
`\__fp_parse_function:NNN`

```
\cs_new:Npn \__fp_parse_unary_function:nNN #1#2#3
  {
    \exp_after:wN \__fp_parse_apply_unary:NNNwN
    \exp_after:wN #3
    \exp_after:wN #2
    \cs:w __fp_#1_o:w \exp_after:wN \cs_end:
    \tex_romannumeral:D
    \__fp_parse_operand:Nw \c_fifteen \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_function:NNN #1#2#3
  {
    \exp_after:wN \__fp_parse_apply_unary:NNNwN
    \exp_after:wN #3
    \exp_after:wN #2
    \exp_after:wN #1
    \tex_romannumeral:D
    \__fp_parse_operand:Nw \c_sixteen \__fp_parse_expand:w
  }
```

(*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`

Those functions are also unary (not binary), but may receive a variable number of arguments.

```
\cs_new_nopar:Npn \__fp_parse_word_acot:N
  { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_i:nn }
\cs_new_nopar:Npn \__fp_parse_word_acotd:N
  { \__fp_parse_function:NNN \__fp_acot_o:Nw \use_ii:nn }
\cs_new_nopar:Npn \__fp_parse_word_atan:N
  { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_i:nn }
\cs_new_nopar:Npn \__fp_parse_word_atand:N
  { \__fp_parse_function:NNN \__fp_atan_o:Nw \use_ii:nn }
\cs_new_nopar:Npn \__fp_parse_word_max:N
  { \__fp_parse_function:NNN \__fp_minmax_o:Nw 2 }
\cs_new_nopar:Npn \__fp_parse_word_min:N
  { \__fp_parse_function:NNN \__fp_minmax_o:Nw 0 }
```

(*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.

```
\cs_new:Npn \__fp_parse_word_abs:N
  { \__fp_parse_unary_function:nNN { set_sign } 0 }
\cs_new_nopar:Npn \__fp_parse_word_exp:N
  { \__fp_parse_unary_function:nNN {exp} ? }
\cs_new_nopar:Npn \__fp_parse_word_ln:N
  { \__fp_parse_unary_function:nNN {ln} ? }
\cs_new_nopar:Npn \__fp_parse_word_sqrt:N
  { \__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.

```
\tl_map_inline:nn
  {
    {acos} {acsc} {asec} {asin}
    {cos} {cot} {csc} {sec} {sin} {tan}
  }
  {
    \cs_new_nopar:cpn { __fp_parse_word_#1:N }
      { \__fp_parse_unary_function:nNN {#1} \use_i:nn }
    \cs_new_nopar:cpn { __fp_parse_word_#1d:N }
      { \__fp_parse_unary_function:nNN {#1} \use_ii:nn }
  }
```

(*End definition for* `\__fp_parse_word_acos:N` *and others.*)

__fp_parse_word_trunc:N
__fp_parse_word_floor:N
__fp_parse_word_ceil:N

```
\cs_new_nopar:Npn \__fp_parse_word_trunc:N
  { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_zero:NNN }
\cs_new_nopar:Npn \__fp_parse_word_floor:N
  { \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_ninf:NNN }
\cs_new_nopar:Npn \__fp_parse_word_ceil:N
  { \__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`.)

__fp_parse_word_round:N
__fp_parse_round:Nw

```
\cs_new:Npn \__fp_parse_word_round:N #1#2
  {
    \if_meaning:w + #2
      \__fp_parse_round:Nw \__fp_round_to_pinf:NNN
    \else:
      \if_meaning:w 0 #2
        \__fp_parse_round:Nw \__fp_round_to_zero:NNN
      \else:
        \if_meaning:w - #2
          \__fp_parse_round:Nw \__fp_round_to_ninf:NNN
        \fi:
      \fi:
    \fi:
    \__fp_parse_function:NNN
```

```
    \__fp_round_o:Nw \__fp_round_to_nearest:NNN #1
  #2
 }
\cs_new:Npn \__fp_parse_round:Nw
  #1 #2 \__fp_round_to_nearest:NNN #3#4 { #2 #1 #3 }
```

(*End definition for* `\__fp_parse_word_round:N` *and* `\__fp_parse_round:Nw`.)

26.6 Main functions

`\__fp_parse:n`
`\__fp_parse_after:ww`

Start a `\romannumeral` 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. Then stop the initial expansion with `\c_zero`.

```
\cs_new:Npn \__fp_parse:n #1
 {
  \tex_romannumeral:D
    \exp_after:wN \__fp_parse_after:ww
    \tex_romannumeral:D
      \__fp_parse_operand:Nw \c_minus_one
      \__fp_parse_expand:w #1 \s__fp_mark
    \s__fp_stop
 }
\cs_new:Npn \__fp_parse_after:ww
  #1@ \__fp_parse_infix_end:N \s__fp_mark \s__fp_stop
 { \c_zero #1 }
```

(*End definition for* `\__fp_parse: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 `\tex_romannumeral:D`. This function should be used with much care.

```
\cs_new:Npn \__fp_parse_operand:Nw #1
 {
  -‘0
  \exp_after:wN \__fp_parse_continue:NwN
  \exp_after:wN #1
  \tex_romannumeral:D -‘0
  \exp_after:wN \__fp_parse_one:Nw
  \exp_after:wN #1
  \tex_romannumeral:D
 }
\cs_new:Npn \__fp_parse_continue:NwN #1 #2 @ #3 { #3 #1 #2 @ }
```

(*End definition for* `\__fp_parse_operand:Nw`.)

`\__fp_parse_apply_binary:NwNwN`

Receives ⟨*precedence*⟩ ⟨*operand*$_1$⟩ `@` ⟨*operation*⟩ ⟨*operand*$_2$⟩ `@` ⟨*infix command*⟩. Builds the appropriate call to the ⟨*operation*⟩ `#3`.

```
\cs_new:Npn \__fp_parse_apply_binary:NwNwN #1 #2@ #3 #4@ #5
 {
  \exp_after:wN \__fp_parse_continue:NwN
  \exp_after:wN #1
```

```
    \tex_romannumeral:D -`0 \cs:w __fp_#3_o:ww \cs_end: #2 #4
    \tex_romannumeral:D -`0 #5 #1
  }
```

(*End definition for* `\__fp_parse_apply_binary:NwNwN`.)

26.7 Infix operators

`\__fp_parse_infix_after_operand:NwN`

```
\cs_new:Npn \__fp_parse_infix_after_operand:NwN #1 #2;
  {
    \__fp_exp_after_f:nw { \__fp_parse_infix:NN #1 }
    #2;
  }
\group_begin:
  \char_set_catcode_letter:N \*
  \cs_new:Npn \__fp_parse_infix:NN #1 #2
    {
      \if_catcode:w \scan_stop: \exp_not:N #2
        \if_int_compare:w
            \pdftex_strcmp:D { \s__fp_mark } { \exp_not:N #2 }
            = \c_zero
          \exp_after:wN \exp_after:wN
          \exp_after:wN \__fp_parse_infix_end:N
        \else:
          \exp_after:wN \exp_after:wN
          \exp_after:wN \__fp_parse_infix_juxtapose:N
        \fi:
      \else:
        \if_int_compare:w
            \__int_eval:w
              ( `#2 \if_int_compare:w `#2 > `Z - \c_thirty_two \fi: )
              / 26
            = \c_three
          \exp_after:wN \exp_after:wN
          \exp_after:wN \__fp_parse_infix_juxtapose:N
        \else:
          \exp_after:wN \__fp_parse_infix_check:NNN
          \cs:w
            __fp_parse_infix_#2:N
            \exp_after:wN \exp_after:wN \exp_after:wN
          \cs_end:
        \fi:
      \fi:
      #1
      #2
    }
  \cs_new:Npn \__fp_parse_infix_check:NNN #1#2#3
    {
      \if_meaning:w \scan_stop: #1
```

```
      \__msg_kernel_expandable_error:nnn
        { kernel } { fp-missing } { * }
      \exp_after:wN \__fp_parse_infix_*:N
      \exp_after:wN #2
      \exp_after:wN #3
    \else:
      \exp_after:wN #1
      \exp_after:wN #2
      \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
    \fi:
  }
\group_end:
```

(*End definition for* `\__fp_parse_infix_after_operand:NwN`.)

26.7.1 Closing parentheses and commas

`\__fp_parse_infix_end:N`

This one is a little bit odd: force every previous operator to end, regardless of the precedence.

```
\cs_new:Npn \__fp_parse_infix_end:N #1
  { @ \use_none:n \__fp_parse_infix_end:N }
```

(*End 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.

```
\group_begin:
  \char_set_catcode_letter:N \)
  \cs_new:Npn \__fp_parse_infix_):N #1
    {
      \if_int_compare:w #1 < \c_zero
        \__msg_kernel_expandable_error:nnn { kernel } { fp-extra } { ) }
        \exp_after:wN \__fp_parse_infix:NN
        \exp_after:wN #1
        \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
      \else:
        \exp_after:wN @
        \exp_after:wN \use_none:n
        \exp_after:wN \__fp_parse_infix_):N
      \fi:
    }
\group_end:
```

(*End definition for* `\__fp_parse_infix_):N`.)

`\__fp_parse_infix_,:N`

```
\group_begin:
  \char_set_catcode_letter:N \,
  \cs_new:Npn \__fp_parse_infix_,:N #1
    {
      \if_int_compare:w #1 > \c_one
        \exp_after:wN @
```

```
        \exp_after:wN \use_none:n
        \exp_after:wN \__fp_parse_infix_,:N
      \else:
        \if_int_compare:w #1 = \c_one
          \exp_after:wN \__fp_parse_infix_comma:w
          \tex_romannumeral:D
        \else:
          \exp_after:wN \__fp_parse_infix_comma_gobble:w
          \tex_romannumeral:D
        \fi:
        \__fp_parse_operand:Nw \c_one
        \exp_after:wN \__fp_parse_expand:w
      \fi:
    }
  \cs_new:Npn \__fp_parse_infix_comma:w #1 @
    { #1 @ \use_none:n }
  \cs_new:Npn \__fp_parse_infix_comma_gobble:w #1 @
    {
      \__msg_kernel_expandable_error:nn { kernel } { fp-extra-comma }
      @ \use_none:n
    }
\group_end:
```

(*End definition for* `\__fp_parse_infix_` *and* `:N`*.*)

26.7.2 Usual infix operators

__fp_parse_infix_+:N
__fp_parse_infix_-:N
__fp_parse_infix_/:N
__fp_parse_infix_mul:N
__fp_parse_infix_and:N
__fp_parse_infix_or:N
__fp_parse_infix_^:N

As described in the "work plan", each infix operator has an associated `\infix` function, a computing function, and precedence, given as arguments to `\__fp_tmp:w`. Using the general mechanism for arithmetic operations. The power operation must be associative in the opposite order from all others. For this, we use two distinct precedences.

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.

```
\group_begin:
⟨*package⟩
  \cs_set_nopar:Npn \+ { }
⟨/package⟩
  \char_set_catcode_other:N \&
  \char_set_catcode_letter:N \^
  \char_set_catcode_letter:N \/
  \char_set_catcode_letter:N \-
  \char_set_catcode_letter:N \+
  \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
    {
      \cs_new:Npn #1 ##1
        {
          \if_int_compare:w ##1 < #3
            \exp_after:wN @
            \exp_after:wN \__fp_parse_apply_binary:NwNwN
```

```
            \exp_after:wN #2
            \tex_romannumeral:D
            \__fp_parse_operand:Nw #4
            \exp_after:wN \__fp_parse_expand:w
          \else:
            \exp_after:wN @
            \exp_after:wN \use_none:n
            \exp_after:wN #1
          \fi:
        }
    }
  \__fp_tmp:w \__fp_parse_infix_^:N   ^ \c_fifteen \c_fourteen
  \__fp_tmp:w \__fp_parse_infix_/:N   / \c_ten     \c_ten
  \__fp_tmp:w \__fp_parse_infix_mul:N * \c_ten     \c_ten
  \__fp_tmp:w \__fp_parse_infix_-:N   - \c_nine    \c_nine
  \__fp_tmp:w \__fp_parse_infix_+:N   + \c_nine    \c_nine
  \__fp_tmp:w \__fp_parse_infix_and:N & \c_five    \c_five
  \__fp_tmp:w \__fp_parse_infix_or:N  | \c_four    \c_four
\group_end:
```

(*End definition for* `\__fp_parse_infix_+:N` *and others.*)

26.7.3 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`.

```
\cs_new:cpn { __fp_parse_infix_(:N } #1
  { \__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 `max(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.

```
\cs_new:Npn \__fp_parse_infix_juxtapose:N #1
  {
    \if_int_compare:w #1 < \c_thirty_two
      \exp_after:wN @
      \exp_after:wN \__fp_parse_apply_juxtapose:NwwN
      \tex_romannumeral:D
      \__fp_parse_operand:Nw \c_thirty_two
      \exp_after:wN \__fp_parse_expand:w
    \else:
      \exp_after:wN @
      \exp_after:wN \use_none:n
      \exp_after:wN \__fp_parse_infix_juxtapose:N
    \fi:
```

```
  }
\cs_new:Npn \__fp_parse_apply_juxtapose:NwwN #1 #2;#3@ #4;#5@
  {
    \if_catcode:w ^ \tl_to_str:n { #3 #5 } ^
    \else:
      \__fp_error:nffn { invalid-ii }
        { \__fp_array_to_clist:n { #2; #3 } }
        { \__fp_array_to_clist:n { #4; #5 } }
        { }
    \fi:
    \__fp_parse_apply_binary:NwNwN #1 #2;@ * #4;@
  }
```

(*End definition for* `\__fp_parse_infix_juxtapose:N` *and* `\__fp_parse_apply_juxtapose:NwwN`.)

26.7.4 Multi-character cases

__fp_parse_infix_*:N

```
\group_begin:
  \char_set_catcode_letter:N ^
  \cs_new:cpn { __fp_parse_infix_*:N } #1#2
    {
      \if:w * \exp_not:N #2
        \exp_after:wN \__fp_parse_infix_^:N
        \exp_after:wN #1
      \else:
        \exp_after:wN \__fp_parse_infix_mul:N
        \exp_after:wN #1
        \exp_after:wN #2
      \fi:
    }
\group_end:
```

(*End definition for* `\__fp_parse_infix_*:N`.)

__fp_parse_infix_|:Nw
__fp_parse_infix_&:Nw

```
\group_begin:
  \char_set_catcode_letter:N \|
  \char_set_catcode_letter:N \&
  \cs_new:Npn \__fp_parse_infix_|:N #1#2
    {
      \if:w | \exp_not:N #2
        \exp_after:wN \__fp_parse_infix_|:N
        \exp_after:wN #1
        \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
      \else:
        \exp_after:wN \__fp_parse_infix_or:N
        \exp_after:wN #1
        \exp_after:wN #2
      \fi:
    }
```

```
  \cs_new:Npn \__fp_parse_infix_&:N #1#2
    {
      \if:w & \exp_not:N #2
        \exp_after:wN \__fp_parse_infix_&:N
        \exp_after:wN #1
        \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
      \else:
        \exp_after:wN \__fp_parse_infix_and:N
        \exp_after:wN #1
        \exp_after:wN #2
      \fi:
    }
\group_end:
```

(*End definition for* `\__fp_parse_infix_|:Nw`*.*)

26.7.5 Ternary operator

__fp_parse_infix_?:N
__fp_parse_infix_::N

```
\group_begin:
  \char_set_catcode_letter:N \?
  \cs_new:Npn \__fp_parse_infix_?:N #1
    {
      \if_int_compare:w #1 < \c_three
        \exp_after:wN @
        \exp_after:wN \__fp_ternary:NwwN
        \tex_romannumeral:D
        \__fp_parse_operand:Nw \c_three
        \exp_after:wN \__fp_parse_expand:w
      \else:
        \exp_after:wN @
        \exp_after:wN \use_none:n
        \exp_after:wN \__fp_parse_infix_?:N
      \fi:
    }
  \cs_new:Npn \__fp_parse_infix_::N #1
    {
      \if_int_compare:w #1 < \c_three
        \__msg_kernel_expandable_error:nnnn
          { kernel } { fp-missing } { ? } { ~for~?: }
        \exp_after:wN @
        \exp_after:wN \__fp_ternary_auxii:NwwN
        \tex_romannumeral:D
        \__fp_parse_operand:Nw \c_two
        \exp_after:wN \__fp_parse_expand:w
      \else:
        \exp_after:wN @
        \exp_after:wN \use_none:n
        \exp_after:wN \__fp_parse_infix_::N
      \fi:
```

```
    }
\group_end:
```

(*End definition for* `\__fp_parse_infix_?:N` *and* `\__fp_parse_infix_::N`.)

26.7.6 Comparisons

__fp_parse_infix_<:N
__fp_parse_infix_=:N
__fp_parse_infix_>:N
__fp_parse_infix_!:N
__fp_parse_excl_error:
__fp_parse_compare:NNNNNNN
__fp_parse_compare_auxi:NNNNNNN
__fp_parse_compare_auxii:NNNNN
__fp_parse_compare_end:NNNNw
__fp_compare:wNNNNw

```
\cs_new:cpn { __fp_parse_infix_<:N } #1
  {
    \__fp_parse_compare:NNNNNNN #1 \c_one
      \c_zero  \c_zero \c_zero \c_zero <
  }
\cs_new:cpn { __fp_parse_infix_=:N } #1
  {
    \__fp_parse_compare:NNNNNNN #1 \c_one
      \c_zero \c_zero \c_zero \c_zero =
  }
\cs_new:cpn { __fp_parse_infix_>:N } #1
  {
    \__fp_parse_compare:NNNNNNN #1 \c_one
      \c_zero \c_zero \c_zero \c_zero >
  }
\cs_new:cpn { __fp_parse_infix_!:N } #1
  {
    \exp_after:wN \__fp_parse_compare:NNNNNNN
    \exp_after:wN #1
    \exp_after:wN \c_zero
    \exp_after:wN \c_one
    \exp_after:wN \c_one
    \exp_after:wN \c_one
    \exp_after:wN \c_one
  }
\cs_new:Npn \__fp_parse_excl_error:
  {
    \__msg_kernel_expandable_error:nnnn
      { kernel } { fp-missing } { = } { ~after~!. }
  }
\cs_new:Npn \__fp_parse_compare:NNNNNNN #1
  {
    \if_int_compare:w #1 < \c_seven
      \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
      \exp_after:wN \__fp_parse_excl_error:
    \else:
      \exp_after:wN @
      \exp_after:wN \use_none:n
      \exp_after:wN \__fp_parse_compare:NNNNNNN
    \fi:
  }
\cs_new:Npn \__fp_parse_compare_auxi:NNNNNNN #1#2#3#4#5#6#7
```

```
  {
    \if_case:w
          \if_catcode:w \scan_stop: \exp_not:N #7
            \c_minus_one
          \else:
            \__int_eval:w `#7 - `< \__int_eval_end:
          \fi:
         \__fp_parse_compare_auxii:NNNNN #2#2#4#5#6
    \or: \__fp_parse_compare_auxii:NNNNN #2#3#2#5#6
    \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#2#6
    \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#5#2
    \else: #1 \__fp_parse_compare_end:NNNNw #3#4#5#6#7
    \fi:
  }
\cs_new:Npn \__fp_parse_compare_auxii:NNNNN #1#2#3#4#5
  {
    \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
    \exp_after:wN \prg_do_nothing:
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN #4
    \exp_after:wN #5
    \tex_romannumeral:D \exp_after:wN \__fp_parse_expand:w
  }
\cs_new:Npn \__fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
  {
    \fi:
    \exp_after:wN @
    \exp_after:wN \__fp_parse_apply_compare:NwNNNNNwN
    \exp_after:wN \c_one_fp
    \exp_after:wN #1
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN #4
    \tex_romannumeral:D
    \__fp_parse_operand:Nw \c_seven \__fp_parse_expand:w #5
  }
\cs_new:Npn \__fp_parse_apply_compare:NwNNNNNwN
    #1 #2@ #3 #4#5#6#7 #8@ #9
  {
    \if_int_odd:w
        \if_meaning:w \c_zero_fp #3
          \c_zero
        \else:
          \if_case:w \__fp_compare_back:ww #8 #2 \exp_stop_f:
            #5 \or: #6 \or: #7 \else: #4
          \fi:
        \fi:
      \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
```

```
      \exp_after:wN \c_one_fp
    \else:
      \exp_after:wN \__fp_parse_apply_compare_aux:NNwN
      \exp_after:wN \c_zero_fp
    \fi:
    #1 #8 #9
  }
\cs_new:Npn \__fp_parse_apply_compare_aux:NNwN #1 #2 #3; #4
  {
    \if_meaning:w \__fp_parse_compare:NNNNNNN #4
      \exp_after:wN \__fp_parse_continue_compare:NNwNN
      \exp_after:wN #1
      \exp_after:wN #2
      \tex_romannumeral:D -`0
      \__fp_exp_after_o:w #3;
      \tex_romannumeral:D -`0
    \else:
      \exp_after:wN \__fp_parse_continue:NwN
      \exp_after:wN #2
      \tex_romannumeral:D -`0
      \exp_after:wN #1
      \tex_romannumeral:D -`0
    \fi:
    #4 #2
  }
\cs_new:Npn \__fp_parse_continue_compare:NNwNN #1#2 #3@ #4#5
  { #4 #2 #3@ #1 }
```

(*End definition for* `\__fp_parse_infix_<:N` *and others.*)

26.8 Candidate: defining new l3fp functions

`\fp_function:Nw`

This relies on the definition of the null character as a prefix operator with very specific semantics.

```
\group_begin:
  \char_set_catcode_other:N \^^@
  \cs_new:Npn \fp_function:Nw #1 { ^^@ ; { \exp_after:wN #1 } }
\group_end:
```

(*End definition for* `\fp_function:Nw`. *This function is documented on page 198.*)

`\fp_new_function:Npn`
`\__fp_function_args:Nwn`

This relies on the definition of the null character as a prefix operator with very specific semantics.

```
\group_begin:
  \char_set_catcode_other:N \^^@
  \cs_new_protected:Npn \fp_new_function:Npn #1#2#
    {
      \cs_new:Npx #1
        {
          ^^@ ;
            {
```

```
                \exp_not:N \exp_after:wN
                \exp_not:N \__fp_function_args:Nwn
                \exp_not:N \exp_after:wN
                \exp_not:c { __fp_user_ \cs_to_str:N #1 }
                \exp_not:N \__int_value:w
                \int_eval:n { \tl_count:n {#2} / \c_two }
                \exp_not:N \exp_after:wN ;
              }
          }
        \cs_new:cpn { __fp_user_ \cs_to_str:N #1 } #2
      }
\group_end:
\cs_new:Npn \__fp_function_args:Nwn #1#2; #3
  {
    \int_compare:nNnTF { \tl_count:n {#3} } = {#2}
      { #1 #3 }
      {
        \__msg_kernel_expandable_error:nnnnn
          { kernel } { fp-num-args } { #1() } {#2} {#2}
        \c_nan_fp
      }
  }
```

(*End definition for* `\fp_new_function:Npn`. *This function is documented on page* 199.)

`\__fp_parse_prefix_^:Nw`

```
\group_begin:
  \char_set_catcode_other:N \^^@
  \cs_new:cpn { __fp_parse_prefix_ ^^@ :Nw } #1 ; #2
    {
      \exp_after:wN \__fp_function_apply:NnwN
      \exp_after:wN #1
      \exp_after:wN { #2 \exp_after:wN }
      \tex_romannumeral:D
        \__fp_parse_operand:Nw \c_sixteen \__fp_parse_expand:w
    }
\group_end:
```

`\__fp_function_apply:NnwN`
`\__fp_function_store:wwNwnn`
`\__fp_function_store_end:wnnn`

```
\cs_new:Npn \__fp_function_apply:NnwN #1#2#3@#4
  {
    \exp_after:wN \__fp_exp_after_array_f:w
      \tex_romannumeral:D -`0
      \__fp_function_store:wwNwnn #3
        \s__fp_mark \__fp_function_store:wwNwnn ;
        \s__fp_mark \__fp_function_store_end:wnnn
      \s__fp_stop { } { } {#2}
    \s__fp_stop
    #4 #1
  }
```

```
\cs_new:Npn \__fp_function_store:wwNwnn
    #1; #2 \s__fp_mark #3#4 \s__fp_stop #5#6
  { #3 #2 \s__fp_mark #3#4 \s__fp_stop { #5 #6 } { { #1; } } }
\cs_new:Npn \__fp_function_store_end:wnnn
    #1 \s__fp_stop #2#3#4
  { \__fp_parse:n { #4 {#2} } }
```

(*End definition for* `\__fp_function_apply:NnwN`*,* `\__fp_function_store:wwNwnn`*, and* `\__fp_function_store_end:wnnn`*.*)

26.9 Messages

```
\__msg_kernel_new:nnn { kernel } { unknown-fp-word }
  { Unknown~fp~word~#1. }
\__msg_kernel_new:nnn { kernel } { fp-missing }
  { Missing~#1~inserted #2. }
\__msg_kernel_new:nnn { kernel } { fp-extra }
  { Extra~#1~ignored. }
\__msg_kernel_new:nnn { kernel } { fp-early-end }
  { Premature~end~in~fp~expression. }
\__msg_kernel_new:nnn { kernel } { fp-after-e }
  { Cannot~use~#1 after~'e'. }
\__msg_kernel_new:nnn { kernel } { fp-missing-number }
  { Missing~number~before~'#1'. }
\__msg_kernel_new:nnn { kernel } { fp-unknown-symbol }
  { Unknown~symbol~#1~ignored. }
\__msg_kernel_new:nnn { kernel } { fp-extra-comma }
  { Unexpected~comma:~extra~arguments~ignored. }
\__msg_kernel_new:nnn { kernel } { fp-num-args }
  { #1~expects~between~#2~and~#3~arguments. }
```

```
⟨/initex | package⟩
```

27 l3fp-logic Implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

27.1 Syntax of internal functions

- `\__fp_compare_npos:nwnw` {⟨*expo*$_1$⟩} ⟨*body*$_1$⟩ ; {⟨*expo*$_2$⟩} ⟨*body*$_2$⟩ ;
- `\__fp_minmax_o:Nw` ⟨*sign*⟩ ⟨*floating point array*⟩
- `\__fp_not_o:w ?` ⟨*floating point array*⟩ (with one floating point number only)
- `\__fp_&_o:ww` ⟨*floating point*⟩ ⟨*floating point*⟩
- `\__fp_|_o:ww` ⟨*floating point*⟩ ⟨*floating point*⟩
- `\__fp_ternary:NwwN`, `\__fp_ternary_auxi:NwwN`, `\__fp_ternary_auxii:NwwN` have to be understood.

27.2 Existence test

\fp_if_exist_p:N
\fp_if_exist_p:c
\fp_if_exist:NTF
\fp_if_exist:cTF

Copies of the cs functions defined in l3basics.

```
\prg_new_eq_conditional:NNn \fp_if_exist:N \cs_if_exist:N { TF , T , F , p }
\prg_new_eq_conditional:NNn \fp_if_exist:c \cs_if_exist:c { TF , T , F , p }
```

(*End definition for* `\fp_if_exist:N` *and* `\fp_if_exist:c`*. These functions are documented on page* ??.)

27.3 Comparison

\fp_compare_p:n
\fp_compare:nTF
__fp_compare_return:w

Within floating point expressions, comparison operators are treated as operations, so we evaluate #1, then compare with 0.

```
\prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }
  {
    \exp_after:wN \__fp_compare_return:w
    \tex_romannumeral:D -`0 \__fp_parse:n {#1}
  }
\cs_new:Npn \__fp_compare_return:w \s__fp \__fp_chk:w #1#2;
  {
    \if_meaning:w 0 #1
      \prg_return_false:
    \else:
      \prg_return_true:
    \fi:
  }
```

(*End definition for* `\fp_compare:n`*. These functions are documented on page 174.*)

\fp_compare_p:nNn
\fp_compare:nNnTF
__fp_compare_aux:wn

Evaluate #1 and #3, using an auxiliary to expand both, and feed the two floating point numbers swapped to __fp_compare_back:ww, defined below. Compare the result with `#2-`=, which is -1 for <, 0 for =, 1 for > and 2 for ?.

```
\prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
  {
    \if_int_compare:w
        \exp_after:wN \__fp_compare_aux:wn
          \tex_romannumeral:D -`0 \__fp_parse:n {#1} {#3}
        = \__int_eval:w `#2 - `= \__int_eval_end:
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
\cs_new:Npn \__fp_compare_aux:wn #1; #2
  {
    \exp_after:wN \__fp_compare_back:ww
      \tex_romannumeral:D -`0 \__fp_parse:n {#2} #1;
  }
```

(*End definition for* `\fp_compare:nNn`*. These functions are documented on page 174.*)

__fp_compare_back:ww
__fp_compare_nan:w

__fp_compare_back:ww ⟨y⟩ ; ⟨x⟩ ;

Expands (in the same way as \int_eval:n) to -1 if $x < y$, 0 if $x = y$, 1 if $x > y$, and 2 otherwise (denoted as $x?y$). If either operand is nan, stop the comparison with __fp_compare_nan:w returning 2. If x is negative, swap the outputs 1 and -1 (*i.e.*, $>$ and $<$); we can henceforth assume that $x \geq 0$. If $y \geq 0$, and they have the same type, either they are normal and we compare them with __fp_compare_npos:nwnw, or they are equal. If $y \geq 0$, but of a different type, the highest type is a larger number. Finally, if $y \leq 0$, then $x > y$, unless both are zero.

```
\cs_new:Npn \__fp_compare_back:ww
    \s__fp \__fp_chk:w #1 #2 #3;
    \s__fp \__fp_chk:w #4 #5 #6;
  {
    \__int_value:w
      \if_meaning:w 3 #1 \exp_after:wN \__fp_compare_nan:w \fi:
      \if_meaning:w 3 #4 \exp_after:wN \__fp_compare_nan:w \fi:
      \if_meaning:w 2 #5 - \fi:
      \if_meaning:w #2 #5
        \if_meaning:w #1 #4
          \if_meaning:w 1 #1
            \__fp_compare_npos:nwnw #6; #3;
          \else:
            0
          \fi:
        \else:
          \if_int_compare:w #4 < #1 - \fi: 1
        \fi:
      \else:
        \if_int_compare:w #1#4 = \c_zero
          0
        \else:
          1
        \fi:
      \fi:
    \exp_stop_f:
  }
\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
__fp_compare_significand:nnnnnnnn

__fp_compare_npos:nwnw {⟨$expo_1$⟩} ⟨$body_1$⟩ ; {⟨$expo_2$⟩} ⟨$body_2$⟩ ;

Within an __int_value:w ... \exp_stop_f: construction, this expands to 0 if the two numbers are equal, -1 if the first is smaller, and 1 if the first is bigger. First compare the exponents: the larger one denotes the larger number. If they are equal, we must compare significands. If both the first 8 digits and the next 8 digits coincide, the numbers are equal. If only the first 8 digits coincide, the next 8 decide. Otherwise, the first 8 digits are compared.

```
\cs_new:Npn \__fp_compare_npos:nwnw #1#2; #3#4;
  {
    \if_int_compare:w #1 = #3 \exp_stop_f:
```

```
      \__fp_compare_significand:nnnnnnnn #2 #4
    \else:
      \if_int_compare:w #1 < #3 - \fi: 1
    \fi:
  }
\cs_new:Npn \__fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
  {
    \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
      \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
        0
      \else:
        \if_int_compare:w #3#4 < #7#8 - \fi: 1
      \fi:
    \else:
      \if_int_compare:w #1#2 < #5#6 - \fi: 1
    \fi:
  }
```

(*End definition for* `\__fp_compare_npos:nwnw`.)

27.4 Floating point expression loops

\fp_do_until:nn
\fp_do_while:nn
\fp_until_do:nn
\fp_while_do:nn

These are quite easy given the above functions. The `do_until` and `do_while` versions execute the body, then test. The `until_do` and `while_do` do it the other way round.

```
\cs_new:Npn \fp_do_until:nn #1#2
  {
    #2
    \fp_compare:nF {#1}
      { \fp_do_until:nn {#1} {#2} }
  }
\cs_new:Npn \fp_do_while:nn #1#2
  {
    #2
    \fp_compare:nT {#1}
      { \fp_do_while:nn {#1} {#2} }
  }
\cs_new:Npn \fp_until_do:nn #1#2
  {
    \fp_compare:nF {#1}
      {
        #2
        \fp_until_do:nn {#1} {#2}
      }
  }
\cs_new:Npn \fp_while_do:nn #1#2
  {
    \fp_compare:nT {#1}
      {
        #2
        \fp_while_do:nn {#1} {#2}
```

```
      }
  }
```

(*End definition for* `\fp_do_until:nn` *and others. These functions are documented on page 176.*)

`\fp_do_until:nNnn`
`\fp_do_while:nNnn`
`\fp_until_do:nNnn`
`\fp_while_do:nNnn`

As above but not using the `nNn` syntax.

```
\cs_new:Npn \fp_do_until:nNnn #1#2#3#4
  {
    #4
    \fp_compare:nNnF {#1} #2 {#3}
      { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
  }
\cs_new:Npn \fp_do_while:nNnn #1#2#3#4
  {
    #4
    \fp_compare:nNnT {#1} #2 {#3}
      { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
  }
\cs_new:Npn \fp_until_do:nNnn #1#2#3#4
  {
    \fp_compare:nNnF {#1} #2 {#3}
      {
        #4
        \fp_until_do:nNnn {#1} #2 {#3} {#4}
      }
  }
\cs_new:Npn \fp_while_do:nNnn #1#2#3#4
  {
    \fp_compare:nNnT {#1} #2 {#3}
      {
        #4
        \fp_while_do:nNnn {#1} #2 {#3} {#4}
      }
  }
```

(*End definition for* `\fp_do_until:nNnn` *and others. These functions are documented on page 175.*)

27.5 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 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`.

```
\cs_new:Npn \__fp_minmax_o:Nw #1#2 @
  {
    \if_meaning:w 0 #1
      \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN \c_one
```

```
    \else:
      \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN \c_minus_one
    \fi:
    #2
    \s__fp \__fp_chk:w 2 #1 \s__fp_exact ;
    \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;
  }
```

(*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.

```
\cs_new:Npn \__fp_minmax_loop:Nww
    #1 \s__fp \__fp_chk:w #2#3; \s__fp \__fp_chk:w #4#5;
  {
    \if_meaning:w 3 #4
      \if_meaning:w 3 #2
        \__fp_minmax_auxi:ww
      \else:
        \__fp_minmax_auxii:ww
      \fi:
    \else:
      \if_int_compare:w
          \__fp_compare_back:ww
            \s__fp \__fp_chk:w #4#5;
            \s__fp \__fp_chk:w #2#3;
          = #1
        \__fp_minmax_auxii:ww
      \else:
        \__fp_minmax_auxi:ww
      \fi:
    \fi:
    \__fp_minmax_loop:Nww #1
      \s__fp \__fp_chk:w #2#3;
      \s__fp \__fp_chk:w #4#5;
  }
```

(*End definition for* `\__fp_minmax_loop:Nww`.)

`\__fp_minmax_auxi:ww`
`\__fp_minmax_auxii:ww`

Keep the first/second number, and remove the other.

```
\cs_new:Npn \__fp_minmax_auxi:ww  #1 \fi: \fi: #2 \s__fp #3 ; \s__fp #4;
  { \fi: \fi: #2 \s__fp #3 ; }
\cs_new:Npn \__fp_minmax_auxii:ww #1 \fi: \fi: #2 \s__fp #3 ;
  { \fi: \fi: #2 }
```

(*End definition for* `\__fp_minmax_auxi:ww` *and* `\__fp_minmax_auxii:ww`.)

__fp_minmax_break_o:w

This function is called from within an `\if_meaning:w` test. Skip to the end of the tests, close the current test with `\fi:`, clean up, and return the appropriate number with one post-expansion.

```
\cs_new:Npn \__fp_minmax_break_o:w #1 \fi: \fi: #2 \s__fp #3; #4;
  { \fi: \__fp_exp_after_o:w \s__fp #3; }
```

(*End definition for* `\__fp_minmax_break_o:w`.)

27.6 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.

```
\cs_new:cpn { __fp_not_o:w } #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_meaning:w 0 #2
      \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
    \else:
      \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
    \fi:
  }
```

(*End definition for* `\__fp_not_o:w`.)

__fp_&_o:ww
__fp_|_o:ww
__fp_and_return:wNw

For `and`, if the first number is zero, return it (with the same sign). Otherwise, return the second one. For `or`, the logic is reversed: if the first number is non-zero, return it, otherwise return the second number: we achieve that by hi-jacking `\__fp_&_o:ww`, inserting an extra argument, `\else:`, before `\s__fp`. In all cases, expand after the floating point number.

```
\group_begin:
  \char_set_catcode_letter:N &
  \char_set_catcode_letter:N |
  \cs_new:Npn \__fp_&_o:ww #1 \s__fp \__fp_chk:w #2#3;
    {
      \if_meaning:w 0 #2 #1
        \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
      \fi:
      \__fp_exp_after_o:w
    }
  \cs_new_nopar:Npn \__fp_|_o:ww { \__fp_&_o:ww \else: }
\group_end:
\cs_new:Npn \__fp_and_return:wNw #1; \fi: #2#3; { \fi: #2 #1; }
```

(*End definition for* `\__fp_&_o:ww`.)

27.7 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.

```
\cs_new:Npn \__fp_ternary:NwwN #1 #2@ #3@ #4
  {
    \if_meaning:w \__fp_parse_infix_::N #4
      \__fp_ternary_loop:Nw
        #2
        \s__fp \__fp_chk:w { \__fp_ternary_loop_break:w } ;
      \__fp_ternary_break_point:n { \exp_after:wN \__fp_ternary_auxi:NwwN }
      \exp_after:wN #1
      \tex_romannumeral:D -`0
      \__fp_exp_after_array_f:w #3 \s__fp_stop
      \exp_after:wN @
      \tex_romannumeral:D
        \__fp_parse_operand:Nw \c_two
        \__fp_parse_expand:w
    \else:
      \__msg_kernel_expandable_error:nnnn
        { kernel } { fp-missing } { : } { ~for~?: }
      \exp_after:wN \__fp_parse_continue:NwN
      \exp_after:wN #1
      \tex_romannumeral:D -`0
      \__fp_exp_after_array_f:w #3 \s__fp_stop
      \exp_after:wN #4
      \exp_after:wN #1
    \fi:
  }
\cs_new:Npn \__fp_ternary_loop_break:w #1 \fi: #2 \__fp_ternary_break_point:n #3
  {
    \c_zero = \c_zero \fi:
    \exp_after:wN \__fp_ternary_auxii:NwwN
  }
\cs_new:Npn \__fp_ternary_loop:Nw \s__fp \__fp_chk:w #1#2;
  {
    \if_int_compare:w #1 > \c_zero
      \exp_after:wN \__fp_ternary_map_break:
    \fi:
    \__fp_ternary_loop:Nw
  }
\cs_new:Npn \__fp_ternary_map_break: #1 \__fp_ternary_break_point:n #2 {#2}
\cs_new:Npn \__fp_ternary_auxi:NwwN #1#2@#3@#4
  {
    \exp_after:wN \__fp_parse_continue:NwN
    \exp_after:wN #1
    \tex_romannumeral:D -`0
    \__fp_exp_after_array_f:w #2 \s__fp_stop
    #4 #1
  }
\cs_new:Npn \__fp_ternary_auxii:NwwN #1#2@#3@#4
```

```
  {
    \exp_after:wN \__fp_parse_continue:NwN
    \exp_after:wN #1
    \tex_romannumeral:D -`0
    \__fp_exp_after_array_f:w #3 \s__fp_stop
    #4 #1
  }
```

(*End definition for* `\__fp_ternary:NwwN`*,* `\__fp_ternary_auxi:NwwN`*, and* `\__fp_ternary_auxii:NwwN`*.*)

```
⟨/initex | package⟩
```

28 l3fp-basics Implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

The l3fp-basics module implements addition, subtraction, multiplication, and division of two floating points, and the absolute value and sign-changing operations on one floating point. All operations implemented in this module yield the outcome of rounding the infinitely precise result of the operation to the nearest floating point.

Some algorithms used below end up being quite similar to some described in "What Every Computer Scientist Should Know About Floating Point Arithmetic", by David Goldberg, which can be found at `http://cr.yp.to/2005-590/goldberg.pdf`.

28.1 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}`.

```
\cs_new:Npn \__fp_basics_pack_low:NNNNNw #1 #2#3#4#5 #6;
  { + #1 - \c_one ; {#2#3#4#5} {#6} ; }
\cs_new:Npn \__fp_basics_pack_high:NNNNNw #1 #2#3#4#5 #6;
  {
    \if_meaning:w 2 #1
      \__fp_basics_pack_high_carry:w
    \fi:
    ; {#2#3#4#5} {#6}
  }
\cs_new:Npn \__fp_basics_pack_high_carry:w \fi: ; #1
  { \fi: + \c_one ; {1000} }
```

(*End definition for* `\__fp_basics_pack_low:NNNNNw`*,* `\__fp_basics_pack_high:NNNNNw`*, and* `\__fp_basics_pack_high_carry:`

`\__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.

```
\cs_new:Npn \__fp_basics_pack_weird_low:NNNNw #1 #2#3#4 #5;
  {
```

```
      \if_meaning:w 2 #1
        + \c_one
      \fi:
      \__int_eval_end:
      #2#3#4; {#5} ;
  }
\cs_new:Npn \__fp_basics_pack_weird_high:NNNNNNNNw
    1 #1#2#3#4 #5#6#7#8 #9; { ; {#1#2#3#4} {#5#6#7#8} {#9} }
```

(*End definition for* `\__fp_basics_pack_weird_low:NNNNw` *and* `\__fp_basics_pack_weird_high:NNNNNNNNw`.)

28.2 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.

28.2.1 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.

```
\cs_new_nopar:cpx { __fp_-_o:ww } \s__fp
  {
```

```
    \exp_not:c { __fp_+_o:ww }
    \exp_not:n { \s__fp \__fp_neg_sign:N }
  }
```

(*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 ⟨*sign*$_2$⟩ 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 ⟨*sign*$_2$⟩ (expansion of `#1#5`) as an argument. If the ⟨*types*⟩ are distinct, the result is simply the floating point number with the highest ⟨*type*⟩. Since case 3 (used for two `nan`) also picks the first operand, we can also use it when ⟨*type*$_1$⟩ is greater than ⟨*type*$_2$⟩. Also note that we don't need to worry about ⟨*sign*$_2$⟩ in that case since the second operand is discarded.

```
\cs_new:cpn { __fp_+_o:ww }
    \s__fp #1 \__fp_chk:w #2 #3 ; \s__fp \__fp_chk:w #4 #5
  {
    \if_case:w
      \if_meaning:w #2 #4
        #2 \exp_stop_f:
      \else:
        \if_int_compare:w #2 > #4 \exp_stop_f:
          \c_three
        \else:
          \c_minus_one
        \fi:
      \fi:
           \exp_after:wN \__fp_add_zeros_o:Nww \__int_value:w
    \or:   \exp_after:wN \__fp_add_normal_o:Nww \__int_value:w
    \or:   \exp_after:wN \__fp_add_inf_o:Nww \__int_value:w
    \or:   \__fp_case_return_i_o:ww
    \else: \exp_after:wN \__fp_add_return_ii_o:Nww \__int_value:w
    \fi:
    #1 #5
    \s__fp \__fp_chk:w #2 #3 ;
    \s__fp \__fp_chk:w #4 #5
  }
```

(*End definition for* `\__fp_+_o:ww`.)

`\__fp_add_return_ii_o:Nww` Ignore the first operand, and return the second, but using the sign `#1` rather than `#4`. As usual, expand after the floating point.

```
\cs_new:Npn \__fp_add_return_ii_o:Nww #1 #2 ; \s__fp \__fp_chk:w #3 #4
  { \__fp_exp_after_o:w \s__fp \__fp_chk:w #3 #1 }
```

(*End definition for* `\__fp_add_return_ii_o:Nww`.)

`\__fp_add_zeros_o:Nww` Adding two zeros yields `\c_zero_fp`, except if both zeros were -0.

```
\cs_new:Npn \__fp_add_zeros_o:Nww #1 \s__fp \__fp_chk:w 0 #2
  {
```

```
    \if_int_compare:w #2 #1 = 20 \exp_stop_f:
      \exp_after:wN \__fp_add_return_ii_o:Nww
    \else:
      \__fp_case_return_i_o:ww
    \fi:
    #1
    \s__fp \__fp_chk:w 0 #2
  }
```

(*End 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 ⟨*sign*$_2$⟩ (#1) and the ⟨*sign*$_2$⟩ (#4) are identical.

```
\cs_new:Npn \__fp_add_inf_o:Nww
    #1 \s__fp \__fp_chk:w 2 #2 #3; \s__fp \__fp_chk:w 2 #4
  {
    \if_meaning:w #1 #2
      \__fp_case_return_i_o:ww
    \else:
      \__fp_case_use:nw
        {
          \if_meaning:w #1 #4
            \exp_after:wN \__fp_invalid_operation_o:Nww
            \exp_after:wN +
          \else:
            \exp_after:wN \__fp_invalid_operation_o:Nww
            \exp_after:wN -
          \fi:
        }
    \fi:
    \s__fp \__fp_chk:w 2 #2 #3;
    \s__fp \__fp_chk:w 2 #4
  }
```

(*End definition for* `\__fp_add_inf_o:Nww`.)

`\__fp_add_normal_o:Nww`

`\__fp_add_normal_o:Nww` ⟨*sign*$_2$⟩ `\s__fp \__fp_chk:w 1` ⟨*sign*$_1$⟩ ⟨*exp*$_1$⟩ ⟨*body*$_1$⟩ `; \s__fp \__fp_chk:w 1` ⟨*initial sign*$_2$⟩ ⟨*exp*$_2$⟩ ⟨*body*$_2$⟩ `;`

We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.

```
\cs_new:Npn \__fp_add_normal_o:Nww #1 \s__fp \__fp_chk:w 1 #2
  {
    \if_meaning:w #1#2
      \exp_after:wN \__fp_add_npos_o:NnwNnw
    \else:
      \exp_after:wN \__fp_sub_npos_o:NnwNnw
    \fi:
    #2
  }
```

(*End definition for* `\__fp_add_normal_o:Nww`.)

28.2.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

__fp_add_npos_o:NnwNnw

`\__fp_add_npos_o:NnwNnw` ⟨*sign*$_1$⟩ ⟨*exp*$_1$⟩ ⟨*body*$_1$⟩ `; \s__fp \__fp_chk:w 1` ⟨*initial sign*$_2$⟩ ⟨*exp*$_2$⟩ ⟨*body*$_2$⟩ `;`

Since we are doing an addition, the final sign is ⟨*sign*$_1$⟩. Start an `\__int_eval:w`, responsible for computing the exponent: the result, and the ⟨*final sign*⟩ are then given to `\__fp_sanitize:Nw` which checks for overflow. The exponent is computed as the largest exponent `#2` or `#5`, incremented if there is a carry. To add the significands, we decimate the smaller number by the difference between the exponents. This is done by `\__fp_add_big_i:wNww` or `\__fp_add_big_ii:wNww`. We need to bring the final sign with us in the midst of the calculation to round properly at the end.

```
\cs_new:Npn \__fp_add_npos_o:NnwNnw #1#2#3 ; \s__fp \__fp_chk:w 1 #4 #5
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN #1
    \int_use:N \__int_eval:w
      \if_int_compare:w #2 > #5 \exp_stop_f:
        #2
        \exp_after:wN \__fp_add_big_i_o:wNww \__int_value:w -
      \else:
        #5
        \exp_after:wN \__fp_add_big_ii_o:wNww \__int_value:w
      \fi:
      \__int_eval:w #5 - #2 ; #1 #3;
  }
```

(*End definition for* `\__fp_add_npos_o:NnwNnw`.)

__fp_add_big_i_o:wNww
__fp_add_big_ii_o:wNww

`\__fp_add_big_i_o:wNww` ⟨*shift*⟩ `;` ⟨*final sign*⟩ ⟨*body*$_1$⟩ `;` ⟨*body*$_2$⟩ `;`

Shift the significand of the small number, then add with `\__fp_add_significand_o:NnnwnnnnN`.

```
\cs_new:Npn \__fp_add_big_i_o:wNww #1; #2 #3; #4;
  {
    \__fp_decimate:nNnnnn {#1}
      \__fp_add_significand_o:NnnwnnnnN
      #4
    #3
    #2
  }
\cs_new:Npn \__fp_add_big_ii_o:wNww #1; #2 #3; #4;
  {
    \__fp_decimate:nNnnnn {#1}
      \__fp_add_significand_o:NnnwnnnnN
      #3
    #4
    #2
  }
```

(*End definition for* `\__fp_add_big_i_o:wNww`.)

__fp_add_significand_o:NnnwnnnnN
__fp_add_significand_pack:NNNNNNN
__fp_add_significand_test_o:N

__fp_add_significand_o:NnnwnnnnN ⟨*rounding digit*⟩ {⟨Y'_1⟩} {⟨Y'_2⟩} ⟨*extra-digits*⟩ ; {⟨X_1⟩} {⟨X_2⟩} {⟨X_3⟩} {⟨X_4⟩} ⟨*final sign*⟩

To round properly, we must know at which digit the rounding should occur. This requires to know whether the addition produces an overall carry or not. Thus, we do the computation now and check for a carry, then go back and do the rounding. The rounding may cause a carry in very rare cases such as $0.99\cdots95 \to 1.00\cdots0$, but this situation always give an exact power of 10, for which it is easy to correct the result at the end.

```
\cs_new:Npn \__fp_add_significand_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
  {
    \exp_after:wN \__fp_add_significand_test_o:N
    \int_use:N \__int_eval:w 1#5#6 + #2
      \exp_after:wN \__fp_add_significand_pack:NNNNNNN
      \int_use:N \__int_eval:w 1#7#8 + #3 ; #1
  }
\cs_new:Npn \__fp_add_significand_pack:NNNNNNN #1 #2#3#4#5#6#7
  {
    \if_meaning:w 2 #1
      + \c_one
    \fi:
    ; #2 #3 #4 #5 #6 #7 ;
  }
\cs_new:Npn \__fp_add_significand_test_o:N #1
  {
    \if_meaning:w 2 #1
      \exp_after:wN \__fp_add_significand_carry_o:wwwNN
    \else:
      \exp_after:wN \__fp_add_significand_no_carry_o:wwwNN
    \fi:
  }
```

(*End definition for* __fp_add_significand_o:NnnwnnnnN.)

__fp_add_significand_no_carry_o:wwwNN

__fp_add_significand_no_carry_o:wwwNN ⟨*8d*⟩ ; ⟨*6d*⟩ ; ⟨*2d*⟩ ; ⟨*rounding digit*⟩ ⟨*sign*⟩

If there's no carry, grab all the digits again and round. The packing function __fp_basics_pack_high:NNNNNw takes care of the case where rounding brings a carry.

```
\cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
    #1; #2; #3#4 ; #5#6
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \int_use:N \__int_eval:w 1 #1
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \int_use:N \__int_eval:w 1 #2 #3#4
        + \__fp_round:NNN #6 #4 #5
        \exp_after:wN ;
  }
```

(*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.

```
\cs_new:Npn \__fp_add_significand_carry_o:wwwNN
    #1; #2; #3#4; #5#6
  {
    + \c_one
    \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw
    \int_use:N \__int_eval:w 1 1 #1
      \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
      \int_use:N \__int_eval:w 1 #2#3 +
        \exp_after:wN \__fp_round:NNN
        \exp_after:wN #6
        \exp_after:wN #3
        \__int_value:w \__fp_round_digit:Nw #4 #5 ;
        \exp_after:wN ;
  }
```

(*End definition for* `\__fp_add_significand_carry_o:wwwNN`.)

28.2.3 Absolute subtraction

__fp_sub_npos_o:NnwNnw
__fp_sub_eq_o:Nnwnw
__fp_sub_npos_ii_o:Nnwnw

`\__fp_sub_npos_o:NnwNnw` ⟨*sign*$_1$⟩ ⟨*exp*$_1$⟩ ⟨*body*$_1$⟩ `; \s__fp \__fp_chk:w 1` ⟨*initial sign*$_2$⟩ ⟨*exp*$_2$⟩ ⟨*body*$_2$⟩ `;`

Rounding properly in some modes requires to know what the sign of the result will be. Thus, we start by comparing the exponents and significands. If the numbers coincide, return zero. If the second number is larger, swap the numbers and call `\__fp_sub_npos_i_o:Nnwnw` with the opposite of ⟨*sign*$_1$⟩.

```
\cs_new:Npn \__fp_sub_npos_o:NnwNnw #1#2#3; \s__fp \__fp_chk:w 1 #4#5#6;
  {
    \if_case:w \__fp_compare_npos:nwnw {#2} #3; {#5} #6; \exp_stop_f:
      \exp_after:wN \__fp_sub_eq_o:Nnwnw
    \or:
      \exp_after:wN \__fp_sub_npos_i_o:Nnwnw
    \else:
      \exp_after:wN \__fp_sub_npos_ii_o:Nnwnw
    \fi:
    #1 {#2} #3; {#5} #6;
  }
\cs_new:Npn \__fp_sub_eq_o:Nnwnw #1#2; #3; { \exp_after:wN \c_zero_fp }
\cs_new:Npn \__fp_sub_npos_ii_o:Nnwnw #1 #2; #3;
  {
    \exp_after:wN \__fp_sub_npos_i_o:Nnwnw
      \int_use:N \__int_eval:w \c_two - #1 \__int_eval_end:
      #3; #2;
  }
```

(*End definition for* `\__fp_sub_npos_o:NnwNnw`.)

__fp_sub_npos_i_o:Nnwnw

After the computation is done, `\__fp_sanitize:Nw` checks for overflow/underflow. It expects the ⟨*final sign*⟩ and the ⟨*exponent*⟩ (delimited by `;`). Start an integer expression for the exponent, which starts with the exponent of the largest number, and may be decreased if the two numbers are very close. If the two numbers have the same exponent, call the `near` auxiliary. Otherwise, decimate y, then call the `far` auxiliary to evaluate the difference between the two significands. Note that we decimate by 1 less than one could expect.

```
\cs_new:Npn \__fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN #1
    \int_use:N \__int_eval:w
      #2
      \if_int_compare:w #2 = #4 \exp_stop_f:
        \exp_after:wN \__fp_sub_back_near_o:nnnnnnnnN
      \else:
        \exp_after:wN \__fp_decimate:nNnnnn \exp_after:wN
          { \int_use:N \__int_eval:w #2 - #4 - \c_one \exp_after:wN }
          \exp_after:wN \__fp_sub_back_far_o:NnnwnnnnN
      \fi:
        #5
      #3
      #1
  }
```

(*End definition for* `\__fp_sub_npos_i_o:Nnwnw`.)

__fp_sub_back_near_o:nnnnnnnnN
__fp_sub_back_near_pack:NNNNNNw
__fp_sub_back_near_after:wNNNNw

`\__fp_sub_back_near_o:nnnnnnnnN` {⟨Y_1⟩} {⟨Y_2⟩} {⟨Y_3⟩} {⟨Y_4⟩} {⟨X_1⟩} {⟨X_2⟩} {⟨X_3⟩} {⟨X_4⟩} ⟨*final sign*⟩

In this case, the subtraction is exact, so we discard the ⟨*final sign*⟩ `#9`. The very large shifts of 10^9 and $1.1 \cdot 10^9$ are unnecessary here, but allow the auxiliaries to be reused later. Each integer expression produces a 10 digit result. If the resulting 16 digits start with a 0, then we need to shift the group, padding with trailing zeros.

```
\cs_new:Npn \__fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
  {
    \exp_after:wN \__fp_sub_back_near_after:wNNNNw
    \int_use:N \__int_eval:w 10#5#6 - #1#2 - \c_eleven
      \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
      \int_use:N \__int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
  }
\cs_new:Npn \__fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
  { + #1#2 ; {#3#4#5#6} {#7} ; }
\cs_new:Npn \__fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 ;
  {
    \if_meaning:w 0 #1
      \exp_after:wN \__fp_sub_back_shift:wnnnn
    \fi:
    ; {#1#2#3#4} {#5}
  }
```

(*End definition for* `\__fp_sub_back_near_o:nnnnnnnnN`.)

__fp_sub_back_shift:wnnnn
__fp_sub_back_shift_ii:ww
__fp_sub_back_shift_iii:NNNNNNNNw
__fp_sub_back_shift_iv:nnnnw

`\__fp_sub_back_shift:wnnnn ; {`$\langle Z_1\rangle$`} {`$\langle Z_2\rangle$`} {`$\langle Z_3\rangle$`} {`$\langle Z_4\rangle$`} ;`

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.

```
\cs_new:Npn \__fp_sub_back_shift:wnnnn ; #1#2
  {
    \exp_after:wN \__fp_sub_back_shift_ii:ww
    \__int_value:w #1 #2 0 ;
  }
\cs_new:Npn \__fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
  {
    \if_meaning:w @ #1 @
      - \c_seven
      - \exp_after:wN \use_i:nnn
        \exp_after:wN \__fp_sub_back_shift_iii:NNNNNNNNw
        \__int_value:w #2#3 0 ~ 123456789;
    \else:
      - \__fp_sub_back_shift_iii:NNNNNNNNw #1 123456789;
    \fi:
    \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
    \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
    \exp_after:wN \__fp_sub_back_shift_iv:nnnnw
    \exp_after:wN ;
    \__int_value:w
    #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
  }
\cs_new:Npn \__fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
\cs_new:Npn \__fp_sub_back_shift_iv:nnnnw #1 ; #2 ; { ; #1 ; }
```

(*End definition for* `\__fp_sub_back_shift:wnnnn`.)

__fp_sub_back_far_o:NnnwnnnnN

`\__fp_sub_back_far_o:NnnwnnnnN` $\langle rounding\rangle$ `{`$\langle Y'_1\rangle$`} {`$\langle Y'_2\rangle$`}` $\langle extra\text{-}digits\rangle$ `; {`$\langle X_1\rangle$`} {`$\langle X_2\rangle$`} {`$\langle X_3\rangle$`} {`$\langle X_4\rangle$`}` $\langle final\ sign\rangle$

If the difference is greater than $10^{\langle expo_x\rangle}$, call the `very_far` auxiliary. If the result is less than $10^{\langle expo_x\rangle}$, call the `not_far` auxiliary. If it is too close a call to know yet, namely if $1\langle Y'_1\rangle\langle Y'_2\rangle = \langle X_1\rangle\langle X_2\rangle\langle X_3\rangle\langle X_4\rangle 0$, then call the `quite_far` auxiliary. We use the odd combination of space and semi-colon delimiters to allow the `not_far` auxiliary to grab each piece individually, the `very_far` auxiliary to use `\__fp_pack_eight:wNNNNNNNN`, and the `quite_far` to ignore the significands easily (using the `;` delimiter).

```
\cs_new:Npn \__fp_sub_back_far_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
  {
    \if_case:w
      \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
        \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
```

```
          \c_zero
        \else:
          \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: \c_one
        \fi:
      \else:
        \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: \c_one
      \fi:
           \exp_after:wN \__fp_sub_back_quite_far_o:wwNN
    \or:   \exp_after:wN \__fp_sub_back_very_far_o:wwwwNN
    \else: \exp_after:wN \__fp_sub_back_not_far_o:wwwwNN
    \fi:
    #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1
  }
```

(*End definition for* `\__fp_sub_back_far_o:NnnwnnnnN`.)

__fp_sub_back_quite_far_o:wwNN
__fp_sub_back_quite_far_ii:NN

The easiest case is when $x - y$ is extremely close to a power of 10, namely the first digit of x is 1, and all others vanish when subtracting y. Then the ⟨*rounding*⟩ `#3` and the ⟨*final sign*⟩ `#4` control whether we get 1 or 0.9999999999999999. In the usual round-to-nearest mode, we 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).

```
\cs_new:Npn \__fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
  {
    \exp_after:wN \__fp_sub_back_quite_far_ii:NN
    \exp_after:wN #3
    \exp_after:wN #4
  }
\cs_new:Npn \__fp_sub_back_quite_far_ii:NN #1#2
  {
    \if_case:w \__fp_round_neg:NNN #2 0 #1
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
      { ; {1000} {0000} {0000} {0000} ; }
      { - \c_one ; {9999} {9999} {9999} {9999} ; }
  }
```

(*End definition for* `\__fp_sub_back_quite_far_o:wwNN`.)

__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`.

```
\cs_new:Npn \__fp_sub_back_not_far_o:wwwwNN #1 ~ #2; #3 ~ #4; #5#6
```

```
  {
    - \c_one
    \exp_after:wN \__fp_sub_back_near_after:wNNNNw
    \int_use:N \__int_eval:w 1#30 - #1 - \c_eleven
      \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
      \int_use:N \__int_eval:w 11 0000 0000 + #40 - #2
        - \exp_after:wN \__fp_round_neg:NNN
          \exp_after:wN #6
          \use_none:nnnnnnn #2 #5
        \exp_after:wN ;
  }
```

(*End definition for* `\__fp_sub_back_not_far_o:wwwwNN`*.*)

`\__fp_sub_back_very_far_o:wwwwNN`
`\__fp_sub_back_very_far_ii_o:nnNwwNN`

The case where $x - y$ and x have the same exponent is a bit more tricky, mostly because it cannot reuse the same auxiliaries. Shift the y significand by adding a leading 0. Then the logic is similar to the `not_far` functions above. Rounding is a bit more complicated: we have two ⟨*rounding*⟩ digits `#3` and `#6` (from the decimation, and from the new shift) to take into account, and getting the parity of the main result requires a computation. The first `\__int_value:w` triggers the second one because the number is unfinished; we can thus not use 0 in place of 2 there.

```
\cs_new:Npn \__fp_sub_back_very_far_o:wwwwNN #1#2#3#4#5#6#7
  {
    \__fp_pack_eight:wNNNNNNNN
    \__fp_sub_back_very_far_ii_o:nnNwwNN
    { 0 #1#2#3 #4#5#6#7 }
    ;
  }
\cs_new:Npn \__fp_sub_back_very_far_ii_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \int_use:N \__int_eval:w 1#4 - #1 - \c_one
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \int_use:N \__int_eval:w 2#5 - #2
        - \exp_after:wN \__fp_round_neg:NNN
          \exp_after:wN #7
          \__int_value:w
            \if_int_odd:w \__int_eval:w #5 - #2 \__int_eval_end:
              1 \else: 2 \fi:
          \__int_value:w \__fp_round_digit:Nw #3 #6 ;
      \exp_after:wN ;
  }
```

(*End definition for* `\__fp_sub_back_very_far_o:wwwwNN`*.*)

28.3 Multiplication

28.3.1 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.

```
\cs_new_nopar:cpn { __fp_*_o:ww }
  {
    \__fp_mul_cases_o:NnNnww
      *
      { - \c_two + }
      \__fp_mul_npos_o:Nww
      { }
  }
```

(*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 $0\times 0 = 0\times 1 = 1\times 0 = 0$ to case 4 or 5 depending on the combined sign, the products $0\times\infty$ and $\infty\times 0$ to case 6 or 7 (invalid operation), and the products $1\times\infty = \infty\times 1 = \infty\times\infty = \infty$ to cases 8 and 9. Note that the code for these two cases (which return $\pm\infty$) is inserted as argument #4, because it differs in the case of divisions.

```
\cs_new:Npn \__fp_mul_cases_o:NnNnww
    #1#2#3#4 \s__fp \__fp_chk:w #5#6#7; \s__fp \__fp_chk:w #8#9
  {
    \if_case:w \__int_eval:w
                 \if_int_compare:w #5 #8 = \c_eleven
                   \c_one
                 \else:
                   \if_meaning:w 3 #8
                     \c_three
                   \else:
                     \if_meaning:w 3 #5
                       \c_two
                     \else:
                       \if_int_compare:w #5 #8 = \c_ten
                         \c_nine #2 - \c_two
                       \else:
                         (#5 #2 #8) / \c_two * \c_two + \c_seven
                       \fi:
                     \fi:
                   \fi:
                 \fi:
                 \if_meaning:w #6 #9 - \c_one \fi:
               \__int_eval_end:
         \__fp_case_use:nw { #3 0 }
```

```
    \or: \__fp_case_use:nw { #3 2 }
    \or: \__fp_case_return_i_o:ww
    \or: \__fp_case_return_ii_o:ww
    \or: \__fp_case_return_o:Nww \c_zero_fp
    \or: \__fp_case_return_o:Nww \c_minus_zero_fp
    \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
    \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
    \or: \__fp_case_return_o:Nww \c_inf_fp
    \or: \__fp_case_return_o:Nww \c_minus_inf_fp
    #4
    \fi:
    \s__fp \__fp_chk:w #5 #6 #7;
    \s__fp \__fp_chk:w #8 #9
  }
```

(*End definition for* `\__fp_mul_cases_o:nNnnww`*.*)

28.3.2 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` ⟨*sign*$_1$⟩ `{`⟨*exp*$_1$⟩`}` ⟨*body*$_1$⟩ `; \s__fp \__fp_chk:w 1` ⟨*sign*$_2$⟩ `{`⟨*exp*$_2$⟩`}` ⟨*body*$_2$⟩ `;`

After the computation, `\__fp_sanitize:Nw` checks for overflow or underflow. As we did for addition, `\__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`.

```
\cs_new:Npn \__fp_mul_npos_o:Nww
    #1 \s__fp \__fp_chk:w #2 #3 #4 #5 ; \s__fp \__fp_chk:w #6 #7 #8 #9 ;
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN #1
    \int_use:N \__int_eval:w
      #4 + #8
      \__fp_mul_significand_o:nnnnNnnnn #5 #1 #9
  }
```

(*End definition for* `\__fp_mul_npos_o:Nww`*.*)

`\__fp_mul_significand_o:nnnnNnnnn`
`\__fp_mul_significand_drop:NNNNNw`
`\__fp_mul_significand_keep:NNNNNw`

`\__fp_mul_significand_o:nnnnNnnnn {`⟨X_1⟩`} {`⟨X_2⟩`} {`⟨X_3⟩`} {`⟨X_4⟩`}` ⟨*sign*⟩ `{`⟨Y_1⟩`} {`⟨Y_2⟩`} {`⟨Y_3⟩`} {`⟨Y_4⟩`}`

Note the three semicolons at the end of the definition. One is for the last `\__fp_-mul_significand_drop:NNNNNw`; one is for `\__fp_round_digit:Nw` later on; and one, preceded by `\exp_after:wN`, which is correctly expanded (within an `\__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`.

```
\cs_new:Npn \__fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
  {
    \exp_after:wN \__fp_mul_significand_test_f:NNN
    \exp_after:wN #5
    \int_use:N \__int_eval:w 99990000 + #1*#6 +
      \exp_after:wN \__fp_mul_significand_keep:NNNNNw
      \int_use:N \__int_eval:w 99990000 + #1*#7 + #2*#6 +
        \exp_after:wN \__fp_mul_significand_keep:NNNNNw
        \int_use:N \__int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
          \exp_after:wN \__fp_mul_significand_drop:NNNNNw
          \int_use:N \__int_eval:w  99990000 + #1*#9 + #2*#8 + #3*#7 + #4*#6 +
            \exp_after:wN \__fp_mul_significand_drop:NNNNNw
            \int_use:N \__int_eval:w  99990000 + #2*#9 + #3*#8 + #4*#7 +
              \exp_after:wN \__fp_mul_significand_drop:NNNNNw
              \int_use:N \__int_eval:w  99990000 + #3*#9 + #4*#8 +
                \exp_after:wN \__fp_mul_significand_drop:NNNNNw
                \int_use:N \__int_eval:w 100000000 + #4*#9 ;
    ; \exp_after:wN ;
  }
\cs_new:Npn \__fp_mul_significand_drop:NNNNNw #1#2#3#4#5 #6;
  { #1#2#3#4#5 ; + #6 }
\cs_new:Npn \__fp_mul_significand_keep:NNNNNw #1#2#3#4#5 #6;
  { #1#2#3#4#5 ; #6 ; }
```

(*End definition for* `\__fp_mul_significand_o:nnnnNnnnn`*.*)

__fp_mul_significand_test_f:NNN

`\__fp_mul_significand_test_f:NNN` ⟨*sign*⟩ `1` ⟨*digits 1–8*⟩ `;` ⟨*digits 9–12*⟩ `;` ⟨*digits 13–16*⟩ `; +` ⟨*digits 17–20*⟩ `+` ⟨*digits 21–24*⟩ `+` ⟨*digits 25–28*⟩ `+` ⟨*digits 29–32*⟩ `; \exp_after:wN ;`

If the ⟨*digit 1*⟩ is non-zero, then for rounding we only care about the digits 16 and 17, and whether further digits are zero or not (check for exact ties). On the other hand, if ⟨*digit 1*⟩ is zero, we care about digits 17 and 18, and whether further digits are zero.

```
\cs_new:Npn \__fp_mul_significand_test_f:NNN #1 #2 #3
  {
    \if_meaning:w 0 #3
      \exp_after:wN \__fp_mul_significand_small_f:NNwwwN
    \else:
      \exp_after:wN \__fp_mul_significand_large_f:NwwNNNN
    \fi:
    #1 #3
  }
```

(*End definition for* `\__fp_mul_significand_test_f:NNN`*.*)

__fp_mul_significand_large_f:NwwNNNN

In this branch, ⟨*digit 1*⟩ is non-zero. The result is thus ⟨*digits 1–16*⟩, plus some rounding which depends on the digits 16, 17, and whether all subsequent digits are zero or not. Here, `\__fp_round_digit:Nw` takes digits 17 and further (as an integer expression), and replaces it by a ⟨*rounding digit*⟩, suitable for `\__fp_round:NNN`.

```
\cs_new:Npn \__fp_mul_significand_large_f:NwwNNNN #1 #2; #3; #4#5#6#7; +
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \int_use:N \__int_eval:w 1#2
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \int_use:N \__int_eval:w 1#3#4#5#6#7
        + \exp_after:wN \__fp_round:NNN
          \exp_after:wN #1
          \exp_after:wN #7
          \__int_value:w \__fp_round_digit:Nw
  }
```

(*End 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.

```
\cs_new:Npn \__fp_mul_significand_small_f:NNwwwN #1 #2#3; #4#5; #6; + #7
  {
    - \c_one
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \int_use:N \__int_eval:w 1#3#4
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \int_use:N \__int_eval:w 1#5#6#7
        + \exp_after:wN \__fp_round:NNN
          \exp_after:wN #1
          \exp_after:wN #7
          \__int_value:w \__fp_round_digit:Nw
  }
```

(*End definition for* `\__fp_mul_significand_small_f:NNwwwN`.)

28.4 Division

28.4.1 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.

```
\cs_new_nopar:cpn { __fp_/_o:ww }
  {
    \__fp_mul_cases_o:NnNnww
      /
```

```
        { - }
        \__fp_div_npos_o:Nww
        {
          \or:
            \__fp_case_use:nw
              { \__fp_division_by_zero_o:NNww \c_inf_fp / }
          \or:
            \__fp_case_use:nw
              { \__fp_division_by_zero_o:NNww \c_minus_inf_fp / }
        }
    }
```

(*End definition for* `\__fp_/_o:ww`.)

`\__fp_div_npos_o:Nww`

`\__fp_div_npos_o:Nww` ⟨*final sign*⟩ `\s__fp \__fp_chk:w 1` ⟨$sign_A$⟩ `{`⟨*exp A*⟩`}` `{`⟨A_1⟩`}` `{`⟨A_2⟩`}` `{`⟨A_3⟩`}` `{`⟨A_4⟩`} ; \s__fp \__fp_chk:w 1` ⟨$sign_Z$⟩ `{`⟨*exp Z*⟩`}` `{`⟨Z_1⟩`}` `{`⟨Z_2⟩`}` `{`⟨Z_3⟩`}` `{`⟨Z_4⟩`} ;`

We want to compute A/Z. As for multiplication, `\__fp_sanitize:Nw` checks for overflow or underflow; we provide it with the ⟨*final sign*⟩, and an integer expression in which we compute the exponent. We set up the arguments of `\__fp_div_significand_i_o:wnnw`, namely an integer ⟨y⟩ obtained by adding 1 to the first 5 digits of Z (explanation given soon below), then the four `{`⟨A_i⟩`}`, then the four `{`⟨Z_i⟩`}`, a semi-colon, and the ⟨*final sign*⟩, used for rounding at the end.

```
\cs_new:Npn \__fp_div_npos_o:Nww
    #1 \s__fp \__fp_chk:w 1 #2 #3 #4 ; \s__fp \__fp_chk:w 1 #5 #6 #7#8#9;
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN #1
    \int_use:N \__int_eval:w
      #3 - #6
      \exp_after:wN \__fp_div_significand_i_o:wnnw
        \int_use:N \__int_eval:w #7 \use_i:nnnn #8 + \c_one ;
        #4
        {#7}{#8}#9 ;
        #1
  }
```

(*End definition for* `\__fp_div_npos_o:Nww`.)

28.4.2 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 `\__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,

$$\begin{aligned} Q_A &= \left\lfloor \frac{A_1 A_2 0}{y} - \frac{1}{2} \right\rfloor \\ &> \frac{A_1 A_2 0}{y} - \frac{3}{2}. \end{aligned}$$

Note that $10^4 < y \leq 10^5$, and $999 \leq Q_A \leq 99989$. Also note that this formula does not cause an overflow as long as $A < (2^{31}-1)/10^9 \simeq 2.147\cdots$, since the numerator involves an integer slightly smaller than $10^9 A$.

Let us bound B:

$$\begin{aligned}10^5 B &= A_1A_20 + 10\cdot 0.A_3A_4 - 10\cdot Z_1.Z_2Z_3Z_4\cdot Q_A\\ &< A_1A_20\cdot\left(1 - 10\cdot\frac{Z_1.Z_2Z_3Z_4}{y}\right) + \frac{3}{2}\cdot 10\cdot Z_1.Z_2Z_3Z_4 + 10\\ &\leq \frac{A_1A_20\cdot(y - 10\cdot Z_1.Z_2Z_3Z_4)}{y} + \frac{3}{2}y + 10\\ &\leq \frac{A_1A_20\cdot 1}{y} + \frac{3}{2}y + 10 \leq \frac{10^9 A}{y} + 1.6\cdot y.\end{aligned}$$

At the last step, we hide 10 into the second term for later convenience. The same reasoning yields

$$\begin{aligned}10^5 B &< 10^9 A/y + 1.6y,\\ 10^5 C &< 10^9 B/y + 1.6y,\\ 10^5 D &< 10^9 C/y + 1.6y,\\ 10^5 E &< 10^9 D/y + 1.6y.\end{aligned}$$

The goal is now to prove that none of B, C, D, and E can go beyond $(2^{31}-1)/10^9 = 2.147\cdots$.

Combining the various inequalities together with $A < 1$, we get

$$\begin{aligned}&10^5 B < 10^9/y + 1.6y,\\ &10^5 C < 10^{13}/y^2 + 1.6(y + 10^4),\\ &10^5 D < 10^{17}/y^3 + 1.6(y + 10^4 + 10^8/y),\\ &10^5 E < 10^{21}/y^4 + 1.6(y + 10^4 + 10^8/y + 10^{12}/y^2).\end{aligned}$$

All of those bounds are convex functions of y (since every power of y involved is convex, and the coefficients are positive), and thus maximal at one of the end-points of the allowed range $10^4 < y \leq 10^5$. Thus,

$$\begin{aligned}10^5 B &< \max(1.16\cdot 10^5, 1.7\cdot 10^5),\\ 10^5 C &< \max(1.32\cdot 10^5, 1.77\cdot 10^5),\\ 10^5 D &< \max(1.48\cdot 10^5, 1.777\cdot 10^5),\\ 10^5 E &< \max(1.64\cdot 10^5, 1.7777\cdot 10^5).\end{aligned}$$

All of those bounds are less than $2.147\cdot 10^5$, and we are thus within TeX's bounds in all cases!

We 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

$$\frac{2E}{Z} = 2\frac{10^5 E}{10^5 Z} \leq 2\frac{10^5 E}{10^4} < 36,$$

this integer part is between 0 and 35 inclusive. We let ε-TeX round

$$P = \texttt{\int\_eval:n} \left\{ \frac{2 \cdot E_1 E_2}{Z_1 Z_2} \right\},$$

which differs from $2E/Z$ by at most

$$\frac{1}{2} + 2\left|\frac{E}{Z} - \frac{E}{10^{-8} Z_1 Z_2}\right| + 2\left|\frac{10^8 E - E_1 E_2}{Z_1 Z_2}\right| < 1,$$

($1/2$ comes from ε-TeX's rounding) because each absolute value is less than 10^{-7}. Thus P is either the correct integer part, or is off by 1; furthermore, if $2E/Z$ is an integer, $P = 2E/Z$. We will check the sign of $2E - PZ$. If it is negative, then $E/Z \in \big((P-1)/2, P/2\big)$. If it is zero, then $E/Z = P/2$. If it is positive, then $E/Z \in \big(P/2, (P-1)/2\big)$. In each case, we know how to round to an integer, depending on the parity of P, and the rounding mode.

28.4.3 Implementing the significand division

__fp_div_significand_i_o:wnnw

`\__fp_div_significand_i_o:wnnw` ⟨y⟩ `;` `{`⟨A_1⟩`}` `{`⟨A_2⟩`}` `{`⟨A_3⟩`}` `{`⟨A_4⟩`}` `{`⟨Z_1⟩`}` `{`⟨Z_2⟩`}` `{`⟨Z_3⟩`}` `{`⟨Z_4⟩`}` `;` ⟨*sign*⟩

Compute $10^6 + Q_A$ (a 7 digit number thanks to the shift), unbrace ⟨A_1⟩ and ⟨A_2⟩, and prepare the ⟨*continuation*⟩ arguments for 4 consecutive calls to `\__fp_div_significand_calc:wwnnnnnnn`. Each of these calls will need ⟨y⟩ (`#1`), and it turns out that we need post-expansion there, hence the `\__int_value:w`. Here, `#4` is six brace groups, which give the six first `n`-type arguments of the `calc` function.

```
\cs_new:Npn \__fp_div_significand_i_o:wnnw #1 ; #2#3 #4 ;
  {
    \exp_after:wN \__fp_div_significand_test_o:w
    \int_use:N \__int_eval:w
      \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
      \int_use:N \__int_eval:w 999999 + #2 #3 0 / #1 ;
```

```
        #2 #3 ;
        #4
        { \exp_after:wN \__fp_div_significand_ii:wwn \__int_value:w #1 }
        { \exp_after:wN \__fp_div_significand_ii:wwn \__int_value:w #1 }
        { \exp_after:wN \__fp_div_significand_ii:wwn \__int_value:w #1 }
        { \exp_after:wN \__fp_div_significand_iii:wwnnnnn \__int_value:w #1 }
  }
```

(*End 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):

$$\begin{aligned} &10^{-4}(\#2 - \#1 \cdot \#5 - 10 \cdot \langle i\rangle \cdot \#5\#6) + 10^{-8}(\#3 - \#1 \cdot \#6 - 10 \cdot \langle i\rangle \cdot \#7) \\ &+ 10^{-12}(\#4 - \#1 \cdot \#7 - 10 \cdot \langle i\rangle \cdot \#8) + 10^{-16}(-\#1 \cdot \#8), \end{aligned}$$

where $\langle i\rangle$ stands for the 10^5 digit of Q_A, which is 0 or 1, and #1, #2, *etc.* are the parameters of either auxiliary. The factors of 10 come from the fact that $Q_A = 10 \cdot 10^4 \cdot \langle i\rangle + \#1$. As usual, to combine all the terms, we need to choose some shifts which must ensure that the number of digits of the second, third, and fourth terms are each fixed. Here, the positive contributions are at most 10^8 and the negative contributions can go up to 10^9. Indeed, for the auxiliary with $\langle i\rangle = 1$, `#1` is at most 80000, leading to contributions of at worse $-8 \cdot 10^8 4$, while the other negative term is very small $< 10^6$ (except in the first expression, where we don't care about the number of digits); for the auxiliary with $\langle i\rangle = 0$, `#1` can go up to 99999, but there is no other negative term. Hence, a good choice is $2 \cdot 10^9$, which produces totals in the range $[10^9, 2.1 \cdot 10^9]$. We are flirting with TeX's limits once more.

```
\cs_new:Npn \__fp_div_significand_calc:wwnnnnnnn 1#1
  {
    \if_meaning:w 1 #1
      \exp_after:wN \__fp_div_significand_calc_i:wwnnnnnnn
    \else:
```

```
      \exp_after:wN \__fp_div_significand_calc_ii:wwnnnnnnn
    \fi:
  }
\cs_new:Npn \__fp_div_significand_calc_i:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
  {
    1 1 #1
    #9 \exp_after:wN ;
    \int_use:N \__int_eval:w \c__fp_Bigg_leading_shift_int
      + #2 - #1 * #5 - #5#60
      \exp_after:wN \__fp_pack_Bigg:NNNNNNw
      \int_use:N \__int_eval:w \c__fp_Bigg_middle_shift_int
        + #3 - #1 * #6 - #70
        \exp_after:wN \__fp_pack_Bigg:NNNNNNw
        \int_use:N \__int_eval:w \c__fp_Bigg_middle_shift_int
          + #4 - #1 * #7 - #80
          \exp_after:wN \__fp_pack_Bigg:NNNNNNw
          \int_use:N \__int_eval:w \c__fp_Bigg_trailing_shift_int
            - #1 * #8 ;
    {#5}{#6}{#7}{#8}
  }
\cs_new:Npn \__fp_div_significand_calc_ii:wwnnnnnnn #1; #2;#3#4 #5#6#7#8 #9
  {
    1 0 #1
    #9 \exp_after:wN ;
    \int_use:N \__int_eval:w \c__fp_Bigg_leading_shift_int
      + #2 - #1 * #5
      \exp_after:wN \__fp_pack_Bigg:NNNNNNw
      \int_use:N \__int_eval:w \c__fp_Bigg_middle_shift_int
        + #3 - #1 * #6
        \exp_after:wN \__fp_pack_Bigg:NNNNNNw
        \int_use:N \__int_eval:w \c__fp_Bigg_middle_shift_int
          + #4 - #1 * #7
          \exp_after:wN \__fp_pack_Bigg:NNNNNNw
          \int_use:N \__int_eval:w \c__fp_Bigg_trailing_shift_int
            - #1 * #8 ;
    {#5}{#6}{#7}{#8}
  }
```

(*End definition for* `\__fp_div_significand_calc:wwnnnnnnn`.)

__fp_div_significand_ii:wwn

`\__fp_div_significand_ii:wwn` ⟨*y*⟩ `;` ⟨B_1⟩ `;` `{`⟨B_2⟩`}` `{`⟨B_3⟩`}` `{`⟨B_4⟩`}` `{`⟨Z_1⟩`}` `{`⟨Z_2⟩`}` `{`⟨Z_3⟩`}` `{`⟨Z_4⟩`}` ⟨*continuations*⟩ ⟨*sign*⟩

Compute Q_B by evaluating $\langle B_1\rangle\langle B_2\rangle 0/y-1$. 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.

```
\cs_new:Npn \__fp_div_significand_ii:wwn #1; #2;#3
  {
    \exp_after:wN \__fp_div_significand_pack:NNN
```

```
    \int_use:N \__int_eval:w
      \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
      \int_use:N \__int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
  }
```

(*End definition for* `\__fp_div_significand_ii:wwn`.)

`\__fp_div_significand_iii:wwnnnnn`

`\__fp_div_significand_iii:wwnnnnn` ⟨*y*⟩ `;` ⟨E_1⟩ `;` `{`⟨E_2⟩`}` `{`⟨E_3⟩`}` `{`⟨E_4⟩`}` `{`⟨Z_1⟩`}` `{`⟨Z_2⟩`}` `{`⟨Z_3⟩`}` `{`⟨Z_4⟩`}` ⟨*sign*⟩

We compute $P \simeq 2E/Z$ by rounding $2E_1E_2/Z_1Z_2$. Note the first 0, which multiplies Q_D by 10: we 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.

```
\cs_new:Npn \__fp_div_significand_iii:wwnnnnn #1; #2;#3#4#5 #6#7
  {
    0
    \exp_after:wN \__fp_div_significand_iv:wwnnnnnnn
    \int_use:N \__int_eval:w (\c_two * #2 #3) / #6 #7 ; % <- P
      #2 ; {#3} {#4} {#5}
      {#6} {#7}
  }
```

(*End definition for* `\__fp_div_significand_iii:wwnnnnn`.)

`\__fp_div_significand_iv:wwnnnnnnn`
`\__fp_div_significand_v:NNw`
`\__fp_div_significand_vi:Nw`

`\__fp_div_significand_iv:wwnnnnnnn` ⟨*P*⟩ `;` ⟨E_1⟩ `;` `{`⟨E_2⟩`}` `{`⟨E_3⟩`}` `{`⟨E_4⟩`}` `{`⟨Z_1⟩`}` `{`⟨Z_2⟩`}` `{`⟨Z_3⟩`}` `{`⟨Z_4⟩`}` ⟨*sign*⟩

This adds to the current expression $(10^7 + 10 \cdot Q_D)$ a contribution of $5 \cdot P + \mathrm{sign}(T)$ with $T = 2E - PZ$. This amounts to adding $P/2$ to Q_D, with an extra ⟨*rounding*⟩ digit. This ⟨*rounding*⟩ digit is 0 or 5 if T does not contribute, *i.e.,* if $0 = T = 2E - PZ$, in other words if $10^{16}A/Z$ is an integer or half-integer. Otherwise it is in the appropriate range, $[1, 4]$ or $[6, 9]$. This is precise enough for rounding purposes (in any mode).

It seems an overkill to compute T exactly as I do here, but I see no faster way right now.

Once more, we need to be careful and show that the calculation $\#1 \cdot \#6\#7$ below does not cause an overflow: naively, P can be up to 35, and $\#6\#7$ up to 10^8, but both cannot happen simultaneously. To show that things are fine, we split in two (non-disjoint) cases.

- For $P < 10$, the product obeys $P \cdot \#6\#7 < 10^8 \cdot P < 10^9$.
- For large $P \geq 3$, the rounding error on P, which is at most 1, is less than a factor of 2, hence $P \leq 4E/Z$. Also, $\#6\#7 \leq 10^8 \cdot Z$, hence $P \cdot \#6\#7 \leq 4E \cdot 10^8 < 10^9$.

Both inequalities could be made tighter if needed.

Note however that $P \cdot \#8\#9$ may overflow, since the two factors are now independent, and the result may reach $3.5 \cdot 10^9$. Thus we compute the two lower levels separately. The rest is standard, except that we use `+` as a separator (ending integer expressions explicitly). T is negative if the first character is `-`, it is positive if the first character is neither `0` nor `-`. It is also positive if the first character is `0` and second argument of `\__fp_div_significand_vi:Nw`, a sum of several terms, is also zero. Otherwise, there was an exact agreement: $T = 0$.

```
\cs_new:Npn \__fp_div_significand_iv:wwnnnnnnn #1; #2;#3#4#5 #6#7#8#9
  {
    + \c_five * #1
    \exp_after:wN \__fp_div_significand_vi:Nw
    \int_use:N \__int_eval:w -20 + 2*#2#3 - #1*#6#7 +
      \exp_after:wN \__fp_div_significand_v:NN
      \int_use:N \__int_eval:w 199980 + 2*#4 - #1*#8 +
        \exp_after:wN \__fp_div_significand_v:NN
        \int_use:N \__int_eval:w 200000 + 2*#5 - #1*#9 ;
  }
\cs_new:Npn \__fp_div_significand_v:NN #1#2 { #1#2 \__int_eval_end: + }
\cs_new:Npn \__fp_div_significand_vi:Nw #1#2;
  {
    \if_meaning:w 0 #1
      \if_int_compare:w \__int_eval:w #2 > \c_zero + \c_one \fi:
    \else:
      \if_meaning:w - #1 - \else: + \fi: \c_one
    \fi:
    ;
  }
```

(*End definition for* `\__fp_div_significand_iv:wwnnnnnnn`*,* `\__fp_div_significand_v:NNw`*, and* `\__fp_div_significand_vi:`

`\__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, $\varepsilon = \operatorname{sign}(T)$ is 0 in case $2E = PZ$, 1 in case $2E > PZ$, which means that P was the correct value, but not with an exact quotient, and -1 if $2E < PZ$, *i.e.*, P was an overestimate. The packing function we define now does nothing special: it removes the 10^6 and carries two digits (for the 10^5's and the 10^4's).

```
\cs_new:Npn \__fp_div_significand_pack:NNN 1 #1 #2 { + #1 #2 ; }
```

(*End 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.

```
\cs_new:Npn \__fp_div_significand_test_o:w 10 #1
  {
    \if_meaning:w 0 #1
      \exp_after:wN \__fp_div_significand_small_o:wwwNNNNwN
    \else:
      \exp_after:wN \__fp_div_significand_large_o:wwwNNNNwN
    \fi:
    #1
  }
```

(*End definition for* `\__fp_div_significand_test_o:w`.)

__fp_div_significand_small_o:wwwNNNNwN

`\__fp_div_significand_small_o:wwwNNNNwN 0` ⟨*4d*⟩ `;` ⟨*4d*⟩ `;` ⟨*4d*⟩ `;` ⟨*5d*⟩ `;` ⟨*final sign*⟩

Standard use of `\__fp_basics_pack_low:NNNNNw` and `\__fp_basics_pack_high:NNNNNw`. We finally get to use the ⟨*final sign*⟩ which has been sitting there for a while.

```
\cs_new:Npn \__fp_div_significand_small_o:wwwNNNNwN
    0 #1; #2; #3; #4#5#6#7#8; #9
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \int_use:N \__int_eval:w 1 #1#2
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \int_use:N \__int_eval:w 1 #3#4#5#6#7
        + \__fp_round:NNN #9 #7 #8
        \exp_after:wN ;
  }
```

(*End definition for* `\__fp_div_significand_small_o:wwwNNNNwN`.)

__fp_div_significand_large_o:wwwNNNNwN

`\__fp_div_significand_large_o:wwwNNNNwN` ⟨*5d*⟩ `;` ⟨*4d*⟩ `;` ⟨*4d*⟩ `;` ⟨*5d*⟩ `;` ⟨*sign*⟩

We know that the final result cannot reach 10, hence `1#1#2`, together with contributions from the level below, cannot reach $2 \cdot 10^9$. For rounding, we build the ⟨*rounding digit*⟩ from the last two of our 18 digits.

```
\cs_new:Npn \__fp_div_significand_large_o:wwwNNNNwN
    #1; #2; #3; #4#5#6#7#8; #9
  {
    + \c_one
    \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw
    \int_use:N \__int_eval:w 1 #1 #2
      \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
      \int_use:N \__int_eval:w 1 #3 #4 #5 #6 +
        \exp_after:wN \__fp_round:NNN
        \exp_after:wN #9
        \exp_after:wN #6
        \__int_value:w \__fp_round_digit:Nw #7 #8 ;
      \exp_after:wN ;
  }
```

(*End definition for* `\__fp_div_significand_large_o:wwwNNNNwN`.)

28.5 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.

```
\cs_new:Npn \__fp_sqrt_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_meaning:w 0 #2 \__fp_case_return_same_o:w \fi:
    \if_meaning:w 2 #3
```

```
      \__fp_case_use:nw { \__fp_invalid_operation_o:nw { sqrt } }
    \fi:
    \if_meaning:w 1 #2 \else: \__fp_case_return_same_o:w \fi:
    \__fp_sqrt_npos_o:w
    \s__fp \__fp_chk:w #2 #3 #4;
  }
```

(*End 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;
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN 0
    \int_use:N \__int_eval:w
      \if_int_odd:w #1 \exp_stop_f:
        \exp_after:wN \__fp_sqrt_npos_auxi_o:wwnnN
      \fi:
      #1 / \c_two
      \__fp_sqrt_Newton_o:wwn 56234133; 0; {#2#3} {#4#5} 0
  }
\cs_new:Npn \__fp_sqrt_npos_auxi_o:wwnnN #1 / \c_two #2; 0; #3#4#5
  {
    ( #1 + \c_one ) / \c_two
    \__fp_pack_eight:wNNNNNNNN
    \__fp_sqrt_npos_auxii_o:wNNNNNNNN
    ;
    0 #3 #4
  }
\cs_new:Npn \__fp_sqrt_npos_auxii_o:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
  { \__fp_sqrt_Newton_o:wwn 17782794; 0; {#1} {#2#3#4#5#6#7#8#9} }
```

(*End definition for* `\__fp_sqrt_npos_o:w`.)

`\__fp_sqrt_Newton_o:wwn`

Newton's method maps $x \mapsto \bigl[(x + [10^8 a_1/x])/2\bigr]$ in each iteration, where $[b/c]$ denotes ε-TeX's division. This division rounds the real number b/c to the closest integer, rounding ties away from zero, hence when c is even, $b/c - 1/2 + 1/c \le [b/c] \le b/c + 1/2$ and when c is odd, $b/c - 1/2 + 1/(2c) \le [b/c] \le b/c + 1/2 - 1/(2c)$. For all c, $b/c - 1/2 + 1/(2c) \le [b/c] \le b/c + 1/2$.

Let us prove that the method converges when implemented with ε-TeX integer division, for any $10^6 \le a_1 < 10^8$ and starting value $10^6 \le x < 10^8$. Using the inequalities above and the arithmetic–geometric inequality $(x+t)/2 \ge \sqrt{xt}$ for $t = 10^8 a_1/x$, we find

$$x' = \left[\frac{x + [10^8 a_1/x]}{2}\right] \ge \frac{x + 10^8 a_1/x - 1/2 + 1/(2x)}{2} \ge \sqrt{10^8 a_1} - \frac{1}{4} + \frac{1}{4x}\,.$$

After any step of iteration, we thus have $\delta = x - \sqrt{10^8 a_1} \geq -0.25 + 0.25 \cdot 10^{-8}$. 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 $[10^7, 10^8]$.

```
\cs_new:Npn \__fp_sqrt_Newton_o:wwn #1; #2; #3
  {
    \if_int_compare:w #1 = #2 \exp_stop_f:
      \exp_after:wN \__fp_sqrt_auxi_o:NNNNwnnN
      \int_use:N \__int_eval:w 9999 9999 +
        \exp_after:wN \__fp_use_none_until_s:w
    \fi:
    \exp_after:wN \__fp_sqrt_Newton_o:wwn
    \int_use:N \__int_eval:w (#1 + #3 * 1 0000 0000 / #1) / \c_two ;
    #1; {#3}
  }
```

(*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, $y < 1$.

```
\cs_new:Npn \__fp_sqrt_auxi_o:NNNNwnnN 1 #1#2#3#4#5;
  {
    \__fp_sqrt_auxii_o:NnnnnnnnN
      \__fp_sqrt_auxiii_o:wnnnnnnnn
      {#1#2#3#4} {#5} {2499} {9988} {7500}
  }
```

(*End 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)

$$10^{4j}z = \Big[\big(\lfloor 10^{4j}(a - y^2)\rfloor - 257\big) \cdot (0.5 \cdot 10^8) \Big/ \lfloor 10^8 y + 1\rfloor\Big].$$

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

$$10^{4j}(\sqrt{a} - y) = \frac{10^{4j}\big(a - y^2 - (\sqrt{a} - y)^2\big)}{2y} \geq \frac{\big\lfloor 10^{4j}(a - y^2)\big\rfloor - 257}{2 \cdot 10^{-8}\lfloor 10^8 y + 1\rfloor} + \frac{1}{2}$$

where we have replaced the bound $10^{4j}(16 \cdot 10^{-2j}) = 256$ by 257 and extracted the corresponding term $1/\big(2 \cdot 10^{-8}\lfloor 10^8 y + 1\rfloor\big) \geq 1/2$. Given that ε-TeX's integer division obeys $[b/c] \leq b/c + 1/2$, we deduce that $10^{4j}z \leq 10^{4j}(\sqrt{a} - y)$, hence $y + z \leq \sqrt{a}$ is an underestimate of $\sqrt{a}$, as claimed. One implementation detail: because the computation involves `-#4*#4 - 2*#3*#5 - 2*#2*#6` which may be as low as $-5 \cdot 10^8$, we need to use the `pack_big` functions, and the `big` shifts.

```
\cs_new:Npn \__fp_sqrt_auxii_o:NnnnnnnnN #1 #2#3#4#5#6 #7#8#9
  {
    \exp_after:wN #1
    \int_use:N \__int_eval:w \c__fp_big_leading_shift_int
```

```
      + #7 - #2 * #2
      \exp_after:wN \__fp_pack_big:NNNNNNw
      \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
        - 2 * #2 * #3
        \exp_after:wN \__fp_pack_big:NNNNNNw
        \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
          + #8 - #3 * #3 - 2 * #2 * #4
          \exp_after:wN \__fp_pack_big:NNNNNNw
          \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
            - 2 * #3 * #4 - 2 * #2 * #5
            \exp_after:wN \__fp_pack_big:NNNNNNw
            \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
              + #9 000 0000 - #4 * #4 - 2 * #3 * #5 - 2 * #2 * #6
              \exp_after:wN \__fp_pack_big:NNNNNNw
              \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
                - 2 * #4 * #5 - 2 * #3 * #6
                \exp_after:wN \__fp_pack_big:NNNNNNw
                \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
                  - #5 * #5 - 2 * #4 * #6
                  \exp_after:wN \__fp_pack_big:NNNNNNw
                  \int_use:N \__int_eval:w
                    \c__fp_big_middle_shift_int
                    - 2 * #5 * #6
                    \exp_after:wN \__fp_pack_big:NNNNNNw
                    \int_use:N \__int_eval:w
                      \c__fp_big_trailing_shift_int
                      - #6 * #6 ;
    % (
    - 257 ) * 5000 0000 / (#2#3 + 1) + 10 0000 0000 ;
    {#2}{#3}{#4}{#5}{#6} {#7}{#8}#9
  }
```

(*End definition for* `\__fp_sqrt_auxii_o:NnnnnnnnN`.)

`\__fp_sqrt_auxiii_o:wnnnnnnnn`
`\__fp_sqrt_auxiv_o:NNNNNw`
`\__fp_sqrt_auxv_o:NNNNNw`
`\__fp_sqrt_auxvi_o:NNNNNw`
`\__fp_sqrt_auxvii_o:NNNNNw`

We receive here the difference $a - y^2 = d = \sum_i d_i \cdot 10^{-4i}$, as $\langle d_2\rangle$ `;` `{`$\langle d_3\rangle$`}` … `{`$\langle d_{10}\rangle$`}`, where each block has 4 digits, except $\langle d_2\rangle$. This function finds the largest $j \le 6$ 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

$$10^{4j} z = \Big[\big(\lfloor 10^{4j}(a-y^2)\rfloor - 257\big)\cdot(0.5\cdot 10^8) \Big/ \lfloor 10^8 y + 1\rfloor\Big]$$

for an estimate of $10^{4j}(\sqrt{a}-y)$. If $d_2 \ge 2$, $j = 3$ and the `auxiv` auxiliary receives $10^{12}z$. If $d_2 \le 1$ but $10^4 d_2 + d_3 \ge 2$, $j = 4$ 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 $10^{12}d_2 + 10^8 d_3 + 10^4 d_4 + d_5 \ge 1$. 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 $2 \cdot 10^8 \le 10^{4(j+1)}(a - y^2)$, hence

$$10^{4j} z \ge \frac{(20000-257)(0.5\cdot 10^8)}{\lfloor 10^8 y + 1\rfloor} \ge (20000-257)\cdot 0.5 > 0 \,.$$

```
\cs_new:Npn \__fp_sqrt_auxiii_o:wnnnnnnnn
    #1; #2#3#4#5#6#7#8#9
  {
    \if_int_compare:w #1 > \c_one
      \exp_after:wN \__fp_sqrt_auxiv_o:NNNNNw
      \int_use:N \__int_eval:w (#1#2 %)
    \else:
      \if_int_compare:w #1#2 > \c_one
        \exp_after:wN \__fp_sqrt_auxv_o:NNNNNw
        \int_use:N \__int_eval:w (#1#2#3 %)
      \else:
        \if_int_compare:w #1#2#3 > \c_one
          \exp_after:wN \__fp_sqrt_auxvi_o:NNNNNw
          \int_use:N \__int_eval:w (#1#2#3#4 %)
        \else:
          \exp_after:wN \__fp_sqrt_auxvii_o:NNNNNw
          \int_use:N \__int_eval:w (#1#2#3#4#5 %)
        \fi:
      \fi:
    \fi:
  }
\cs_new:Npn \__fp_sqrt_auxiv_o:NNNNNw 1#1#2#3#4#5#6;
  { \__fp_sqrt_auxviii_o:nnnnnnn {#1#2#3#4#5#6} {00000000} }
\cs_new:Npn \__fp_sqrt_auxv_o:NNNNNw 1#1#2#3#4#5#6;
  { \__fp_sqrt_auxviii_o:nnnnnnn {000#1#2#3#4#5} {#60000} }
\cs_new:Npn \__fp_sqrt_auxvi_o:NNNNNw 1#1#2#3#4#5#6;
  { \__fp_sqrt_auxviii_o:nnnnnnn {0000000#1} {#2#3#4#5#6} }
\cs_new:Npn \__fp_sqrt_auxvii_o:NNNNNw 1#1#2#3#4#5#6;
  {
    \if_int_compare:w #1#2 = \c_zero
      \exp_after:wN \__fp_sqrt_auxx_o:Nnnnnnnn
    \fi:
    \__fp_sqrt_auxviii_o:nnnnnnn {00000000} {000#1#2#3#4#5}
  }
```

(*End definition for* `\__fp_sqrt_auxiii_o:wnnnnnnnn` *and others.*)

`\__fp_sqrt_auxviii_o:nnnnnnn`
`\__fp_sqrt_auxix_o:wnwnw`

Simply add the two 8-digit blocks of z, aligned to the last four of the five 4-digit blocks of y, then call the `auxii` auxiliary to evaluate $y'^2 = (y + z)^2$.

```
\cs_new:Npn \__fp_sqrt_auxviii_o:nnnnnnn #1#2 #3#4#5#6#7
  {
    \exp_after:wN \__fp_sqrt_auxix_o:wnwnw
    \int_use:N \__int_eval:w #3
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \int_use:N \__int_eval:w #1 + 1#4#5
```

```
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \int_use:N \__int_eval:w #2 + 1#6#7 ;
  }
\cs_new:Npn \__fp_sqrt_auxix_o:wnwnw #1; #2#3; #4#5;
  {
    \__fp_sqrt_auxii_o:NnnnnnnnN
      \__fp_sqrt_auxiii_o:wnnnnnnnn {#1}{#2}{#3}{#4}{#5}
  }
```

(*End definition for* `\__fp_sqrt_auxviii_o:nnnnnnn` *and* `\__fp_sqrt_auxix_o:wnwnw`.)

`\__fp_sqrt_auxx_o:Nnnnnnnn`
`\__fp_sqrt_auxxi_o:wwnnN`

At this stage, $j = 6$ and $10^{24}z < 10^7$, hence

$$10^7 + 1/2 > 10^{24}z + 1/2 \geq \bigl(10^{24}(a - y^2) - 258\bigr) \cdot (0.5 \cdot 10^8) \Big/ (10^8 y + 1)\,,$$

then $10^{24}(a - y^2) - 258 < 2(10^7 + 1/2)(y + 10^{-8})$, and

$$10^{24}(a - y^2) < (10^7 + 1290.5)(1 + 10^{-8}/y)(2y) < (10^7 + 1290.5)(1 + 10^{-7})(y + \sqrt{a})\,,$$

which finally implies $0 \leq \sqrt{a} - y < 0.2 \cdot 10^{-16}$. In particular, y is an underestimate of $\sqrt{a}$ and $y + 0.5 \cdot 10^{-16}$ is a (strict) overestimate. There is at exactly one multiple m of $0.5 \cdot 10^{-16}$ in the interval $[y, y + 0.5 \cdot 10^{-16})$. If $m^2 > a$, then the square root is inexact and is obtained by rounding $m - \epsilon$ to a multiple of 10^{-16} (the precise shift $0 < \epsilon < 0.5 \cdot 10^{-16}$ is irrelevant for rounding). If $m^2 = a$ then the square root is exactly m, and there is no rounding. If $m^2 < a$ then we round $m + \epsilon$. For now, discard a few irrelevant arguments `#1`, `#2`, `#3`, and find the multiple of $0.5 \cdot 10^{-16}$ within $[y, y + 0.5 \cdot 10^{-16})$; rather, only the last 4 digits `#8` of y are considered, and we do not perform any carry yet. The `auxxi` auxiliary sets up `auxii` with a continuation function `auxxii` instead of `auxiii` as before. To prevent `auxii` from giving a negative results $a - m^2$, we compute $a + 10^{-16} - m^2$ instead, always positive since $m < \sqrt{a} + 0.5 \cdot 10^{-16}$ and $a \leq 1 - 10^{-16}$.

```
\cs_new:Npn \__fp_sqrt_auxx_o:Nnnnnnnn #1#2#3 #4#5#6#7#8
  {
    \exp_after:wN \__fp_sqrt_auxxi_o:wwnnN
    \int_use:N \__int_eval:w
      (#8 + 2499) / 5000 * 5000 ;
      {#4} {#5} {#6} {#7} ;
  }
\cs_new:Npn \__fp_sqrt_auxxi_o:wwnnN #1; #2; #3#4#5
  {
    \__fp_sqrt_auxii_o:NnnnnnnnN
      \__fp_sqrt_auxxii_o:nnnnnnnnw
      #2 {#1}
      {#3} { #4 + \c_one } #5
  }
```

(*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.

```
\cs_new:Npn \__fp_sqrt_auxxii_o:nnnnnnnnw 0; #1#2#3#4#5#6#7#8 #9;
  {
    \if_int_compare:w #1#2 > \c_zero
      \if_int_compare:w #1#2 = \c_one
        \if_int_compare:w #3#4 = \c_zero
          \if_int_compare:w #5#6 = \c_zero
            \if_int_compare:w #7#8 = \c_zero
              \__fp_sqrt_auxxiii_o:w
            \fi:
          \fi:
        \fi:
      \fi:
      \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnN
      \__int_value:w 9998
    \else:
      \exp_after:wN \__fp_sqrt_auxxiv_o:wnnnnnnnN
      \__int_value:w 10000
    \fi:
    ;
  }
\cs_new:Npn \__fp_sqrt_auxxiii_o:w \fi: \fi: \fi: \fi: #1 \fi: ;
  {
    \fi: \fi: \fi: \fi: \fi:
    \__fp_sqrt_auxxiv_o:wnnnnnnnN 9999 ;
  }
```

(*End definition for* `\__fp_sqrt_auxxii_o:nnnnnnnnw` *and* `\__fp_sqrt_auxxiii_o:w`*.*)

__fp_sqrt_auxxiv_o:wnnnnnnnN

This receives 9998, 9999 or 10000 as `#1` when m is an underestimate, exact, or an overestimate, respectively. Then comes m as five blocks of 4 digits, but where the last block `#6` may be 0, 5000, or 10000. In the latter case, we need to add a carry, unless m is an overestimate (`#1` is then 10000). Then comes a as three arguments. Rounding is done by `\__fp_round:NNN`, whose first argument is the final sign 0 (square roots are positive). We fake its second argument. It should be the last digit kept, but this is only used when ties are “rounded to even”, and only when the result is exactly half-way between two representable numbers rational square roots of numbers with 16 significant digits have: this situation never arises for the square root, as any exact square root of a 16 digit number has at most 8 significant digits. Finally, the last argument is the next digit, possibly shifted by 1 when there are further nonzero digits. This is achieved by `\__fp_round_digit:Nw`, which receives (after removal of the 10000’s digit) one of 0000, 0001, 4999, 5000, 5001, or 9999, which it converts to 0, 1, 4, 5, 6, and 9, respectively.

```
\cs_new:Npn \__fp_sqrt_auxxiv_o:wnnnnnnnN #1; #2#3#4#5#6 #7#8#9
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \int_use:N \__int_eval:w 1 0000 0000 + #2#3
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
```

```
      \int_use:N \__int_eval:w 1 0000 0000
        + #4#5
        \if_int_compare:w #6 > #1 \exp_stop_f: + \c_one \fi:
        + \exp_after:wN \__fp_round:NNN
          \exp_after:wN 0
          \exp_after:wN 0
          \__int_value:w
            \exp_after:wN \use_i:nn
            \exp_after:wN \__fp_round_digit:Nw
            \int_use:N \__int_eval:w #6 + 19999 - #1 ;
    \exp_after:wN ;
  }
```

(*End definition for* `\__fp_sqrt_auxxiv_o:wnnnnnnnN`.)

28.6 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`.

```
\cs_new:Npn \__fp_set_sign_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \exp_after:wN \__fp_exp_after_o:w
    \exp_after:wN \s__fp
    \exp_after:wN \__fp_chk:w
    \exp_after:wN #2
    \__int_value:w
      \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
    #4;
  }
```

(*End definition for* `\__fp_set_sign_o:w`.)

```
⟨/initex | package⟩
```

29 l3fp-extended implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

29.1 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 $\langle$*calculation*$\rangle$ on the two $\langle$*operands*$\rangle$, then feed the result (6 brace groups followed by a semicolon) to the $\langle$*continuation*$\rangle$, responsible for the next step of the calculation. Some functions only accept an `N`-type $\langle$*continuation*$\rangle$. This allows constructions such as

__fp_fixed_add:wwn $\langle X_1\rangle$; $\langle X_2\rangle$;
__fp_fixed_mul:wwn $\langle X_3\rangle$;
__fp_fixed_add:wwn $\langle X_4\rangle$;

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.

29.2 Helpers for numbers with extended precision

`\c__fp_one_fixed_tl` The fixed-point number 1, used in l3fp-expo.

```
\tl_const:Nn \c__fp_one_fixed_tl
  { {10000} {0000} {0000} {0000} {0000} {0000} }
```

(*End 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 $2^{31}-1$).

```
\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 $a_1 \leq 2^{31} - 10001$.

```
\cs_new:Npn \__fp_fixed_add_one:wN #1#2; #3
  {
    \exp_after:wN #3 \exp_after:wN
      { \int_use:N \__int_eval:w \c_ten_thousand + #1 } #2 ;
  }
```

(*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 $[0, 5 \cdot 10^8 - 1]$.

```
\cs_new:Npn \__fp_fixed_div_myriad:wn #1#2#3#4#5#6;
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \int_use:N \__int_eval:w \c__fp_leading_shift_int
      \exp_after:wN \__fp_pack:NNNNNw
      \int_use:N \__int_eval:w \c__fp_trailing_shift_int
        + #1 ; {#2}{#3}{#4}{#5};
  }
```

(*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 ⟨*continuation*⟩ #2 in front. The ⟨*continuation*⟩ was brought up through the expansions by the packing functions.

```
\cs_new:Npn \__fp_fixed_mul_after:wwn #1; #2; #3 { #3 {#1} #2; }
```

(*End definition for* `\__fp_fixed_mul_after:wwn`.)

29.3 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 ⟨*continuation*⟩ {$\langle c_1 \rangle$} ... {$\langle c_6 \rangle$} ; 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).

```
\cs_new:Npn \__fp_fixed_mul_short:wwn #1#2#3#4#5#6; #7#8#9;
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \int_use:N \__int_eval:w \c__fp_leading_shift_int
      + #1*#7
      \exp_after:wN \__fp_pack:NNNNNw
      \int_use:N \__int_eval:w \c__fp_middle_shift_int
        + #1*#8 + #2*#7
        \exp_after:wN \__fp_pack:NNNNNw
        \int_use:N \__int_eval:w \c__fp_middle_shift_int
          + #1*#9 + #2*#8 + #3*#7
          \exp_after:wN \__fp_pack:NNNNNw
          \int_use:N \__int_eval:w \c__fp_middle_shift_int
            + #2*#9 + #3*#8 + #4*#7
            \exp_after:wN \__fp_pack:NNNNNw
            \int_use:N \__int_eval:w \c__fp_middle_shift_int
              + #3*#9 + #4*#8 + #5*#7
              \exp_after:wN \__fp_pack:NNNNNw
              \int_use:N \__int_eval:w \c__fp_trailing_shift_int
                + #4*#9 + #5*#8 + #6*#7
```

```
                + ( #5*#9 + #6*#8 + #6*#9 / \c_ten_thousand )
                / \c_ten_thousand ; ;
  }
```

(*End definition for* `\__fp_fixed_mul_short:wwn`.)

29.4 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 ⟨*a*⟩ by the (small) integer $0 < \langle n\rangle < 10^4$ and feeds the result to the ⟨*continuation*⟩. There is no bound on a_1.

The arguments of the `i` auxiliary are 1: one of the a_i, 2: n, 3: the `ii` or the `iii` auxiliary. It computes a (somewhat tight) lower bound Q_i for the ratio a_i/n.

The `ii` auxiliary receives Q_i, n, and a_i as arguments. It adds Q_i to a surrounding integer expression, and starts a new one with the initial value 9999, which ensures that the result of this expression 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` ⟨*continuation*⟩
$-1 + Q_1$
`\__fp_fixed_div_int_pack:Nw` $9999 + Q_2$
`\__fp_fixed_div_int_pack:Nw` $9999 + Q_3$
`\__fp_fixed_div_int_pack:Nw` $9999 + Q_4$
`\__fp_fixed_div_int_pack:Nw` $9999 + Q_5$
`\__fp_fixed_div_int_pack:Nw` 9999
`\__fp_fixed_div_int_auxii:wnn` Q_6 `;` `{`⟨*n*⟩`}` `{`⟨a_6⟩`}`

where expansion is happening from the last line up. The `iii` auxiliary adds $Q_6 + 2 \simeq a_6/n + 1$ to the last 9999, giving the integer closest to $10000 + a_6/n$.

Each `pack` auxiliary receives 5 digits followed by a semicolon. The first digit is added as a carry to the integer expression above, and the 4 other digits are braced. Each call to the `pack` auxiliary thus produces one brace group. The last brace group is produced by the `after` auxiliary, which places the ⟨*continuation*⟩ as appropriate.

```
\cs_new:Npn \__fp_fixed_div_int:wwN #1#2#3#4#5#6 ; #7 ; #8
  {
    \exp_after:wN \__fp_fixed_div_int_after:Nw
    \exp_after:wN #8
    \int_use:N \__int_eval:w \c_minus_one
      \__fp_fixed_div_int:wnN
      #1; {#7} \__fp_fixed_div_int_auxi:wnn
      #2; {#7} \__fp_fixed_div_int_auxi:wnn
      #3; {#7} \__fp_fixed_div_int_auxi:wnn
      #4; {#7} \__fp_fixed_div_int_auxi:wnn
      #5; {#7} \__fp_fixed_div_int_auxi:wnn
      #6; {#7} \__fp_fixed_div_int_auxii:wnn ;
  }
\cs_new:Npn \__fp_fixed_div_int:wnN #1; #2 #3
  {
```

```
    \exp_after:wN #3
    \int_use:N \__int_eval:w #1 / #2 - \c_one ;
    {#2}
    {#1}
  }
\cs_new:Npn \__fp_fixed_div_int_auxi:wnn #1; #2 #3
  {
    + #1
    \exp_after:wN \__fp_fixed_div_int_pack:Nw
    \int_use:N \__int_eval:w 9999
      \exp_after:wN \__fp_fixed_div_int:wnN
      \int_use:N \__int_eval:w #3 - #1*#2 \__int_eval_end:
  }
\cs_new:Npn \__fp_fixed_div_int_auxii:wnn #1; #2 #3 { + #1 + \c_two ; }
\cs_new:Npn \__fp_fixed_div_int_pack:Nw #1 #2; { + #1; {#2} }
\cs_new:Npn \__fp_fixed_div_int_after:Nw #1 #2; { #1 {#2} }
```

(*End definition for* `\__fp_fixed_div_int:wwN`.)

29.5 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 $0 \le a_1, b_1 \le 114748$, its result must be positive (this happens automatically for addition) and its first group must have at most 5 digits: $(a \pm b)_1 < 100000$. The two functions only differ by a sign, hence use a common auxiliary. It would be nice to grab the 12 brace groups in one go; only 9 parameters are allowed. Start by grabbing the sign, $a_1, \ldots, a_4$, the rest of a, and b_1 and b_2. The second auxiliary receives the rest of a, the sign multiplying b, the rest of b, and the ⟨*continuation*⟩ as arguments. After going down through the various level, we go back up, packing digits and bringing the ⟨*continuation*⟩ (`#8`, then `#7`) from the end of the argument list to its start.

```
\cs_new_nopar:Npn \__fp_fixed_add:wwn { \__fp_fixed_add:Nnnnnwnn + }
\cs_new_nopar:Npn \__fp_fixed_sub:wwn { \__fp_fixed_add:Nnnnnwnn - }
\cs_new:Npn \__fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6; #7#8
  {
    \exp_after:wN \__fp_fixed_add_after:NNNNNwn
    \int_use:N \__int_eval:w 9 9999 9998 + #2#3 #1 #7#8
      \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
      \int_use:N \__int_eval:w 1 9999 9998 + #4#5
        \__fp_fixed_add:nnNnnnwn #6 #1
  }
\cs_new:Npn \__fp_fixed_add:nnNnnnwn #1#2 #3 #4#5 #6#7 ; #8
  {
    #3 #4#5
    \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
    \int_use:N \__int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
  }
\cs_new:Npn \__fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6; #7
  { + #1 ; {#7} {#2#3#4#5} {#6} }
\cs_new:Npn \__fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6; #7
```

```
  { #7 {#1#2#3#4#5} {#6} }
```

(*End definition for* `\__fp_fixed_add:wwn` *and* `\__fp_fixed_sub:wwn`.)

29.6 Multiplying fixed points

`\__fp_fixed_mul:wwn`
`\__fp_fixed_mul:nnnnnnnw`

Computes $a \times b$ and feeds the result to ⟨*continuation*⟩. This function requires $0 \leq a_1, b_1 < 10000$. 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

$$\begin{aligned} a \times b = & a_1 \cdot b_1 \cdot 10^{-8} \\ & + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\ & + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1) \cdot 10^{-16} \\ & + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\ & + \Bigg(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} + a_1 \cdot b_5 + a_5 \cdot b_1 \end{aligned}$$

where the $O(10^{-24})$ stands for terms which are at most $5 \cdot 10^{-24}$; ignoring those leads to an error of at most 5 `ulp`. Note how the first 15 terms only depend on $a_1, \ldots, a_4$ and $b_1, \ldots, b_4$, while the last 6 terms only depend on a_1, a_2, a_5, a_6, and the corresponding parts of b. Hence, the first function grabs $a_1, \ldots, a_4$, the rest of a, and $b_1, \ldots, b_4$, and writes the 15 first terms of the expression, including a left parenthesis for the fraction. The `i` auxiliary receives a_5, a_6, b_1, b_2, a_1, a_2, b_5, b_6 and finally the ⟨*continuation*⟩ as arguments. It writes the end of the expression, including the right parenthesis and the denominator of the fraction. The ⟨*continuation*⟩ is finally placed in front of the 6 brace groups by `\__fp_fixed_mul_after:wwn`.

```
\cs_new:Npn \__fp_fixed_mul:wwn #1#2#3#4 #5; #6#7#8#9
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \int_use:N \__int_eval:w \c__fp_leading_shift_int
      \exp_after:wN \__fp_pack:NNNNNw
      \int_use:N \__int_eval:w \c__fp_middle_shift_int
        + #1*#6
        \exp_after:wN \__fp_pack:NNNNNw
        \int_use:N \__int_eval:w \c__fp_middle_shift_int
          + #1*#7 + #2*#6
          \exp_after:wN \__fp_pack:NNNNNw
          \int_use:N \__int_eval:w \c__fp_middle_shift_int
            + #1*#8 + #2*#7 + #3*#6
            \exp_after:wN \__fp_pack:NNNNNw
            \int_use:N \__int_eval:w \c__fp_middle_shift_int
              + #1*#9 + #2*#8 + #3*#7 + #4*#6
              \exp_after:wN \__fp_pack:NNNNNw
              \int_use:N \__int_eval:w \c__fp_trailing_shift_int
                + #2*#9 + #3*#8 + #4*#7
                + ( #3*#9 + #4*#8
```

```
                  + \__fp_fixed_mul:nnnnnnnw #5 {#6}{#7}  {#1}{#2}
  }
\cs_new:Npn \__fp_fixed_mul:nnnnnnnw #1#2 #3#4 #5#6 #7#8 ;
  {
    #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c_ten_thousand
    + #1*#3 + #5*#7 ; ;
  }
```

(*End definition for* `\__fp_fixed_mul:wwn`.)

29.7 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 ⟨*continuation*⟩. Those functions require $0 \leq a_1, b_1, c_1 \leq 10000$. Since those functions are at the heart of the computation of Taylor expansions, we over-optimize them a bit, and in particular we do not factor out the common parts of the three functions.

For definiteness, consider the task of computing $a \times b + c$. We will perform carries in

$$
\begin{aligned}
a \times b + c = & (a_1 \cdot b_1 + c_1c_2) \cdot 10^{-8} \\
& + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
& + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3c_4) \cdot 10^{-16} \\
& + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
& + \Big(a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} + a_1 \cdot b_5 + a_5
\end{aligned}
$$

where c_1c_2, c_3c_4, c_5c_6 denote the 8-digit number obtained by juxtaposing the two blocks of digits of c, and $\cdot$ denotes multiplication. The task is obviously tough because we have 18 brace groups in front of us.

Each of the three function starts the first two levels (the first, corresponding to 10^{-4}, is empty), with c_1c_2 in the first level, calls the `i` auxiliary with arguments described later, and adds a trailing $+c_5c_6$ `;` `{`⟨*continuation*⟩`}` `;`. The $+c_5c_6$ piece, which is omitted for `\__fp_fixed_one_minus_mul:wwn`, will be taken in the integer expression for the 10^{-24} level.

```
\cs_new:Npn \__fp_fixed_mul_add:wwwn #1; #2; #3#4#5#6#7#8;
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \int_use:N \__int_eval:w \c__fp_big_leading_shift_int
      \exp_after:wN \__fp_pack_big:NNNNNNw
      \int_use:N \__int_eval:w \c__fp_big_middle_shift_int + #3 #4
        \__fp_fixed_mul_add:Nwnnnwnnn +
          + #5 #6 ; #2 ; #1 ; #2 ; +
          + #7 #8 ; ;
  }
\cs_new:Npn \__fp_fixed_mul_sub_back:wwwn #1; #2; #3#4#5#6#7#8;
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \int_use:N \__int_eval:w \c__fp_big_leading_shift_int
      \exp_after:wN \__fp_pack_big:NNNNNNw
```

```
      \int_use:N \__int_eval:w \c__fp_big_middle_shift_int + #3 #4
        \__fp_fixed_mul_add:Nwnnnwnnn -
          + #5 #6 ; #2 ; #1 ; #2 ; -
          + #7 #8 ; ;
  }
\cs_new:Npn \__fp_fixed_one_minus_mul:wwn #1; #2;
  {
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \int_use:N \__int_eval:w \c__fp_big_leading_shift_int
      \exp_after:wN \__fp_pack_big:NNNNNNw
      \int_use:N \__int_eval:w \c__fp_big_middle_shift_int + 1 0000 0000
        \__fp_fixed_mul_add:Nwnnnwnnn -
          ; #2 ; #1 ; #2 ; -
          ; ;
  }
```

(*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.

```
\cs_new:Npn \__fp_fixed_mul_add:Nwnnnwnnn #1 #2; #3#4#5#6; #7#8#9
  {
    #1 #7*#3
    \exp_after:wN \__fp_pack_big:NNNNNNw
    \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
      #1 #7*#4 #1 #8*#3
      \exp_after:wN \__fp_pack_big:NNNNNNw
      \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
        #1 #7*#5 #1 #8*#4 #1 #9*#3 #2
        \exp_after:wN \__fp_pack_big:NNNNNNw
        \int_use:N \__int_eval:w \c__fp_big_middle_shift_int
          #1 \__fp_fixed_mul_add:nnnnwnnnn {#7}{#8}{#9}
  }
```

(*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⟩.

```
\cs_new:Npn \__fp_fixed_mul_add:nnnnwnnnn #1#2#3#4#5; #6#7#8#9
  {
    ( #1*#9 + #2*#8 + #3*#7 + #4*#6 )
    \exp_after:wN \__fp_pack_big:NNNNNNw
    \int_use:N \__int_eval:w \c__fp_big_trailing_shift_int
      \__fp_fixed_mul_add:nnnnwnnwN
        { #6 + #4*#7 + #3*#8 + #2*#9 + #1 }
        { #7 + #4*#8 + #3*#9 + #2 }
        {#1} #5;
        {#6}
  }
```

(*End definition for* `\__fp_fixed_mul_add:nnnnwnnnn`.)

`\__fp_fixed_mul_add:nnnnwnnwN`

Complete the ⟨*partial*$_1$⟩ and ⟨*partial*$_2$⟩ expressions as explained for the `ii` auxiliary. The second one is divided by 10000: this is the carry from level 10^{-28}. The trailing $+c_5c_6$ is taken into the expression for level 10^{-24}. Note that the total of level 10^{-24} is in the interval $[-5\cdot 10^8, 6\cdot 10^8$ (give or take a couple of 10000), hence adding it to the shift gives a 10-digit number, as expected by the packing auxiliaries. See l3fp-aux for the definition of the shifts and packing auxiliaries.

```
\cs_new:Npn \__fp_fixed_mul_add:nnnnwnnwN #1#2 #3#4#5; #6#7#8; #9
  {
    #9 (#4* #1 *#7)
    #9 (#5*#6+#4* #2 *#7+#3*#8) / \c_ten_thousand
  }
```

(*End definition for* `\__fp_fixed_mul_add:nnnnwnnwN`.)

29.8 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`
`\__fp_ep_to_fixed_auxi:www`
`\__fp_ep_to_fixed_auxii:nnnnnnnwn`

Converts an extended-precision number with an exponent at most 4 to a fixed point number whose first block will have 12 digits, most often starting with many zeros.

```
\cs_new:Npn \__fp_ep_to_fixed:wwn #1,#2
  {
    \exp_after:wN \__fp_ep_to_fixed_auxi:www
    \int_use:N \__int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
    \tex_romannumeral:D -`0
    \prg_replicate:nn { \c_four - \int_max:nn {#1} { -32 } } { 0 } ;
  }
\cs_new:Npn \__fp_ep_to_fixed_auxi:www 1#1; #2; #3#4#5#6#7;
  {
    \__fp_pack_eight:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
```

```
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_ep_to_fixed_auxii:nnnnnnnwn ;
    #2 #1#3#4#5#6#7 0000 !
  }
\cs_new:Npn \__fp_ep_to_fixed_auxii:nnnnnnnwn #1#2#3#4#5#6#7; #8! #9
  { #9 {#1#2}{#3}{#4}{#5}{#6}{#7}; }
```

(*End definition for* `\__fp_ep_to_fixed:wwn`.)

__fp_ep_to_ep:wwN
__fp_ep_to_ep_loop:N
__fp_ep_to_ep_end:www
__fp_ep_to_ep_zero:ww

Normalize an extended-precision number. More precisely, leading zeros are removed from the mantissa of the argument, decreasing its exponent as appropriate. Then the digits are packed into 6 groups of 4 (discarding any remaining digit, not rounding). Finally, the continuation `#8` is placed before the resulting exponent–mantissa pair. The input exponent may in fact be given as an integer expression. The `loop` auxiliary grabs a digit: if it is 0, decrement the exponent and continue looping, and otherwise call the `end` auxiliary, which places all digits in the right order (the digit that was not 0, and any remaining digits), followed by some 0, then packs them up neatly in $3 \times 2 = 6$ blocks of four. At the end of the day, remove with `\__fp_use_i:ww` any digit that did not make it in the final mantissa (typically only zeros, unless the original first block has more than 4 digits).

```
\cs_new:Npn \__fp_ep_to_ep:wwN #1,#2#3#4#5#6#7; #8
  {
    \exp_after:wN #8
    \int_use:N \__int_eval:w #1 + \c_four
      \exp_after:wN \use_i:nn
      \exp_after:wN \__fp_ep_to_ep_loop:N
      \int_use:N \__int_eval:w 1 0000 0000 + #2 \__int_eval_end:
      #3#4#5#6#7 ; ; !
  }
\cs_new:Npn \__fp_ep_to_ep_loop:N #1
  {
    \if_meaning:w 0 #1
      - \c_one
    \else:
      \__fp_ep_to_ep_end:www #1
    \fi:
    \__fp_ep_to_ep_loop:N
  }
\cs_new:Npn \__fp_ep_to_ep_end:www
    #1 \fi: \__fp_ep_to_ep_loop:N #2; #3!
  {
    \fi:
    \if_meaning:w ; #1
      - \c_two * \c__fp_max_exponent_int
      \__fp_ep_to_ep_zero:ww
    \fi:
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
```

```
    \__fp_use_i:ww , ;
    #1 #2 0000 0000 0000 0000 0000 0000 ;
  }
\cs_new:Npn \__fp_ep_to_ep_zero:ww \fi: #1; #2; #3;
  { \fi: , {1000}{0000}{0000}{0000}{0000}{0000} ; }
```

(*End definition for* `\__fp_ep_to_ep:wwN`.)

`\__fp_ep_compare:wwww`
`\__fp_ep_compare_aux:wwww`

In l3fp-trig we need to compare two extended-precision numbers. This is based on the same function for positive floating point numbers, with an extra test if comparing only 16 decimals is not enough to distinguish the numbers. Note that this function only works if the numbers are normalized so that their first block is in $[1000, 9999]$.

```
\cs_new:Npn \__fp_ep_compare:wwww #1,#2#3#4#5#6#7;
  { \__fp_ep_compare_aux:wwww {#1}{#2}{#3}{#4}{#5}; #6#7; }
\cs_new:Npn \__fp_ep_compare_aux:wwww #1;#2;#3,#4#5#6#7#8#9;
  {
    \if_case:w
      \__fp_compare_npos:nwnw #1; {#3}{#4}{#5}{#6}{#7}; \exp_stop_f:
            \if_int_compare:w #2 = #8#9 \exp_stop_f:
              0
            \else:
              \if_int_compare:w #2 < #8#9 - \fi: 1
            \fi:
    \or:    1
    \else: -1
    \fi:
  }
```

(*End definition for* `\__fp_ep_compare:wwww`.)

`\__fp_ep_mul:wwwwn`
`\__fp_ep_mul_raw:wwwwN`

Multiply two extended-precision numbers: first normalize them to avoid losing too much precision, then multiply the mantissas `#2` and `#4` as fixed point numbers, and sum the exponents `#1` and `#3`. The result's first block is in $[100, 9999]$.

```
\cs_new:Npn \__fp_ep_mul:wwwwn #1,#2; #3,#4;
  {
    \__fp_ep_to_ep:wwN #3,#4;
    \__fp_fixed_continue:wn
    {
      \__fp_ep_to_ep:wwN #1,#2;
      \__fp_ep_mul_raw:wwwwN
    }
    \__fp_fixed_continue:wn
  }
\cs_new:Npn \__fp_ep_mul_raw:wwwwN #1,#2; #3,#4; #5
  {
    \__fp_fixed_mul:wwn #2; #4;
    { \exp_after:wN #5 \int_use:N \__int_eval:w #1 + #3 , }
  }
```

(*End definition for* `\__fp_ep_mul:wwwwn` *and* `\__fp_ep_mul_raw:wwwwN`.)

29.9 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 $\left[\frac{\bullet}{\bullet}\right]$ denotes ε-TEX's rounding division, which rounds ties away from zero. The idea is to interpolate between $10^3\alpha$ and $10^3\beta$ with a parameter $\langle d_2\rangle/10^4$, so that when $\langle d_2\rangle = 0$ one gets $a = 10^3\beta - 1250 \simeq 10^{12}/\langle d_1\rangle \simeq 10^8/\langle d\rangle$, while when $\langle d_2\rangle = 9999$ one gets $a = 10^3\alpha - 1250 \simeq 10^{12}/(\langle d_1\rangle + 1) \simeq 10^8/\langle d\rangle$. The shift by 1250 helps to ensure that a is an underestimate of the correct value. We 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

$$\frac{\langle n\rangle}{\langle d\rangle} = \frac{\langle n\rangle a}{10^8}\big((1+\epsilon)(1+\epsilon^2)+\epsilon^4\big).$$

Let us prove the upper bound first (multiplied by 10^{15}). Note that $10^7\langle d\rangle < 10^3\langle d_1\rangle + 10^{-1}(\langle d_2\rangle+1)$, and that ε-TEX's division $\left[\frac{\langle d_2\rangle}{10}\right]$ 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,

$$10^7\langle d\rangle a < \left(10^3\langle d_1\rangle + \left[\frac{\langle d_2\rangle}{10}\right] + \frac{1}{2}\right)\left(\left(10^3 - \left[\frac{\langle d_2\rangle}{10}\right]\right)\beta + \left[\frac{\langle d_2\rangle}{10}\right]\alpha - 1250\right) \tag{1}$$
$$< \left(10^3\langle d_1\rangle + \left[\frac{\langle d_2\rangle}{10}\right] + \frac{1}{2}\right)\left(\left(10^3 - \left[\frac{\langle d_2\rangle}{10}\right]\right)\left(\frac{10^9}{\langle d_1\rangle} + \frac{1}{2}\right) + \left[\frac{\langle d_2\rangle}{10}\right]\left(\frac{10^9}{\langle d_1\rangle+1} + \frac{1}{2}\right) - 1250\right) \tag{2}$$
$$< \left(10^3\langle d_1\rangle + \left[\frac{\langle d_2\rangle}{10}\right] + \frac{1}{2}\right)\left(\frac{10^{12}}{\langle d_1\rangle} - \left[\frac{\langle d_2\rangle}{10}\right]\frac{10^9}{\langle d_1\rangle(\langle d_1\rangle+1)} - 750\right) \tag{3}$$

We recognize a quadratic polynomial in $[\langle d_2\rangle/10]$ with a negative leading coefficient: this polynomial is bounded above, according to $([\langle d_2\rangle/10]+a)(b-c[\langle d_2\rangle/10]) \leq (b+ca)^2/(4c)$. Hence,

$$10^7\langle d\rangle a < \frac{10^{15}}{\langle d_1\rangle(\langle d_1\rangle+1)}\left(\langle d_1\rangle + \frac{1}{2} + \frac{1}{4}10^{-3} - \frac{3}{8}\cdot 10^{-9}\langle d_1\rangle(\langle d_1\rangle+1)\right)^2$$

Since $\langle d_1\rangle$ takes integer values within $[1000, 9999]$, it is a simple programming exercise to check that the squared expression is always less than $\langle d_1\rangle(\langle d_1\rangle+1)$, hence $10^7\langle d\rangle a < 10^{15}$. The upper bound is proven. We also find that $\frac{3}{8}$ can be replaced by slightly smaller numbers, but nothing less than $0.374563\ldots$, and going back through the derivation of the upper bound, we find that 1250 is as small a shift as we can obtain without breaking the bound.

Now, the lower bound. The same computation as for the upper bound implies

$$10^7\langle d\rangle a > \left(10^3\langle d_1\rangle + \left[\frac{\langle d_2\rangle}{10}\right] - \frac{1}{2}\right)\left(\frac{10^{12}}{\langle d_1\rangle} - \left[\frac{\langle d_2\rangle}{10}\right]\frac{10^9}{\langle d_1\rangle(\langle d_1\rangle+1)} - 1750\right)$$

This time, we want to find the minimum of this quadratic polynomial. Since the leading coefficient is still negative, the minimum is reached for one of the extreme values $[y/10] = 0$ or $[y/10] = 100$, and we easily check the bound for those values.

We have proven that the algorithm will give us a precise enough answer. Incidentally, the upper bound that we derived tells us that $a < 10^8/\langle d\rangle \leq 10^9$, hence we can compute a safely as a TeX integer, and even add 10^9 to it to ease grabbing of all the digits. The lower bound implies $10^8 - 1755 < a$, which we do not care about.

`\__fp_ep_div:wwwwn` Compute the ratio of two extended-precision numbers. The result is an extended-precision number whose first block lies in the range $[100, 9999]$, and is placed after the ⟨*continuation*⟩ once we are done. First normalize the inputs so that both first block lie in $[1000, 9999]$, then call `\__fp_ep_div_esti:wwwwn` ⟨*denominator*⟩ ⟨*numerator*⟩, responsible for estimating the inverse of the denominator.

```
\cs_new:Npn \__fp_ep_div:wwwwn #1,#2; #3,#4;
  {
    \__fp_ep_to_ep:wwN #1,#2;
    \__fp_fixed_continue:wn
    {
      \__fp_ep_to_ep:wwN #3,#4;
      \__fp_ep_div_esti:wwwwn
    }
  }
```

(*End 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 $\langle n\rangle/(10\langle d\rangle)$). Then the `estii` function evaluates $10^9 + a$, and puts the exponent `#2` after the continuation `#7`: from there on we can forget exponents and focus on the mantissa. The `estiii` function multiplies the denominator `#7` by $10^{-8}a$ (obtained as a

split into the single digit #1 and two blocks of 4 digits, #2#3#4#5 and #6). The result $10^{-8}a\langle d\rangle = (1-\epsilon)$, and a partially packed $10^{-9}a$ (as a block of four digits, and five individual digits, not packed by lack of available macro parameters here) are passed to `\__fp_ep_div_epsi:wnNNNNn`, which computes $10^{-9}a/(1-\epsilon)$, that is, $1/(10\langle d\rangle)$ and we finally multiply this by the numerator #8.

```
\cs_new:Npn \__fp_ep_div_esti:wwwwn #1,#2#3; #4,
  {
    \exp_after:wN \__fp_ep_div_estii:wwnnwwn
    \int_use:N \__int_eval:w 10 0000 0000 / ( #2 + \c_one )
      \exp_after:wN ;
    \int_use:N \__int_eval:w #4 - #1 + \c_one ,
    {#2} #3;
  }
\cs_new:Npn \__fp_ep_div_estii:wwnnwwn #1; #2,#3#4#5; #6; #7
  {
    \exp_after:wN \__fp_ep_div_estiii:NNNNNwwwn
    \int_use:N \__int_eval:w 10 0000 0000 - 1750
      + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
    {#3}{#4}#5; #6; { #7 #2, }
  }
\cs_new:Npn \__fp_ep_div_estiii:NNNNNwwwn 1#1#2#3#4#5#6; #7;
  {
    \__fp_fixed_mul_short:wwn #7; {#1}{#2#3#4#5}{#6};
    \__fp_ep_div_epsi:wnNNNNNn {#1#2#3#4}#5#6
    \__fp_fixed_mul:wwn
  }
```

(*End definition for* `\__fp_ep_div_esti:wwwwn`*,* `\__fp_ep_div_estii:wwnnwwn`*, and* `\__fp_ep_div_estiii:NNNNNwwwn`*.*)

`\__fp_ep_div_epsi:wnNNNNNn`
`\__fp_ep_div_eps_pack:NNNNNw`
`\__fp_ep_div_epsii:wwnNNNNNn`

The bounds shown above imply that the `epsi` function's first operand is $(1-\epsilon)$ with $\epsilon \in [0, 1.755 \cdot 10^{-5}]$. The `epsi` function computes ϵ as $1-(1-\epsilon)$. Since $\epsilon < 10^{-4}$, its first block vanishes and there is no need to explicitly use #1 (which is 9999). Then `epsii` evaluates $10^{-9}a/(1-\epsilon)$ as $(1+\epsilon^2)(1+\epsilon)(10^{-9}a\epsilon) + 10^{-9}a$. Importantly, we compute $10^{-9}a\epsilon$ before multiplying it with the rest, rather than multiplying by ϵ and then $10^{-9}a$, as this second option loses more precision. Also, the combination of `short_mul` and `div_myriad` is both faster and more precise than a simple `mul`.

```
\cs_new:Npn \__fp_ep_div_epsi:wnNNNNNn #1#2#3#4#5#6;
  {
    \exp_after:wN \__fp_ep_div_epsii:wwnNNNNNn
    \int_use:N \__int_eval:w 1 9998 - #2
      \exp_after:wN \__fp_ep_div_eps_pack:NNNNNw
      \int_use:N \__int_eval:w 1 9999 9998 - #3#4
        \exp_after:wN \__fp_ep_div_eps_pack:NNNNNw
        \int_use:N \__int_eval:w 2 0000 0000 - #5#6 ; ;
  }
\cs_new:Npn \__fp_ep_div_eps_pack:NNNNNw #1#2#3#4#5#6;
  { + #1 ; {#2#3#4#5} {#6} }
\cs_new:Npn \__fp_ep_div_epsii:wwnNNNNNn 1#1; #2; #3#4#5#6#7#8
  {
```

```
    \__fp_fixed_mul:wwn {0000}{#1}#2; {0000}{#1}#2;
    \__fp_fixed_add_one:wN
    \__fp_fixed_mul:wwn {10000} {#1} #2 ;
    {
      \__fp_fixed_mul_short:wwn {0000}{#1}#2; {#3}{#4#5#6#7}{#8000};
      \__fp_fixed_div_myriad:wn
      \__fp_fixed_mul:wwn
    }
    \__fp_fixed_add:wwn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000};
  }
```

(*End definition for* `\__fp_ep_div_epsi:wnNNNNNn`*,* `\__fp_ep_div_eps_pack:NNNNNw`*, and* `\__fp_ep_div_epsii:wwnNNNNNn`*.*)

29.10 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 $z \mapsto z(1.5 - 0.5yz^2)$, and we perform a fixed number of steps. Our final result combines r with $y^{-1/2}$ as $x^{-1/2} = 10^{-2} r y^{-1/2}$.

`\__fp_ep_isqrt:wwn`
`\__fp_ep_isqrt_aux:wwn`
`\__fp_ep_isqrt_auxii:wwnnnwn`

First normalize the input, then check the parity of the exponent `#1`. If it is even, the result's exponent will be $-$`#1`$/2$, otherwise it will be $($`#1`$-1)/2$ (except in the case where the input was an exact power of 100). The `auxii` function receives as `#1` the result's exponent just computed, as `#2` the starting value for the iteration giving r (the values 168 and 535 lead to the least number of iterations before convergence, on average), as `#3` and `#4` one empty argument and one `0`, depending on the parity of the original exponent, as `#5` and `#6` the normalized mantissa (`#5` $\in [1000, 9999]$), and as `#7` the continuation. It sets up the iteration giving r: the `esti` function thus receives the initial two guesses `#2` and 0, an approximation `#5` of $10^4 x$ (its first block of digits), and the empty/zero arguments `#3` and `#4`, followed by the mantissa and an altered continuation where we have stored the result's exponent.

```
\cs_new:Npn \__fp_ep_isqrt:wwn #1,#2;
  {
    \__fp_ep_to_ep:wwN #1,#2;
    \__fp_ep_isqrt_auxi:wwn
  }
\cs_new:Npn \__fp_ep_isqrt_auxi:wwn #1,
  {
    \exp_after:wN \__fp_ep_isqrt_auxii:wwnnnwn
```

```
    \int_use:N \__int_eval:w
      \int_if_odd:nTF {#1}
        { (\c_one - #1) / \c_two , 535 , { 0 } { } }
        { \c_one - #1 / \c_two , 168 , { } { 0 } }
  }
\cs_new:Npn \__fp_ep_isqrt_auxii:wwnnnwn #1, #2, #3#4 #5#6; #7
  {
    \__fp_ep_isqrt_esti:wwwnnwn #2, 0, #5, {#3} {#4}
      {#5} #6 ; { #7 #1 , }
  }
```

(*End definition for* `\__fp_ep_isqrt:wwn`.)

`\__fp_ep_isqrt_esti:wwwnnwn`
`\__fp_ep_isqrt_estii:wwwnnwn`
`\__fp_ep_isqrt_estiii:NNNNNwwwn`

If the last two approximations gave the same result, we are done: call the `estii` function to clean up. Otherwise, evaluate $(\langle prev\rangle + 1.005 \cdot 10^{8 \text{ or } 9}/(\langle prev\rangle \cdot x))/2$, 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 $100r^2/2$ or $10r^2/2$ (when the exponent is even or odd, respectively) and feed that to `estiii`. This third auxiliary finds $y_{\text{even}}/2 = 10^{-4}r^2x/2$ or $y_{\text{odd}}/2 = 10^{-5}r^2x/2$ (again, depending on earlier parity). A simple program shows that $y \in [1, 1.0201]$. The number $y/2$ is fed to `\__fp_ep_isqrt_epsi:wN`, which computes $1/\sqrt{y}$, and we finally multiply the result by r.

```
\cs_new:Npn \__fp_ep_isqrt_esti:wwwnnwn #1, #2, #3, #4
  {
    \if_int_compare:w #1 = #2 \exp_stop_f:
      \exp_after:wN \__fp_ep_isqrt_estii:wwwnnwn
    \fi:
    \exp_after:wN \__fp_ep_isqrt_esti:wwwnnwn
    \int_use:N \__int_eval:w
      (#1 + 1 0050 0000 #4 / (#1 * #3)) / \c_two ,
    #1, #3, {#4}
  }
\cs_new:Npn \__fp_ep_isqrt_estii:wwwnnwn #1, #2, #3, #4#5
  {
    \exp_after:wN \__fp_ep_isqrt_estiii:NNNNNwwwn
    \int_use:N \__int_eval:w 1000 0000 + #2 * #2 #5 * \c_five
      \exp_after:wN , \int_use:N \__int_eval:w 10000 + #2 ;
  }
\cs_new:Npn \__fp_ep_isqrt_estiii:NNNNNwwwn 1#1#2#3#4#5#6, 1#7#8; #9;
  {
    \__fp_fixed_mul_short:wwn #9; {#1} {#2#3#4#5} {#600} ;
    \__fp_ep_isqrt_epsi:wN
    \__fp_fixed_mul_short:wwn {#7} {#80} {0000} ;
  }
```

(*End 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.

```
\cs_new:Npn \__fp_ep_isqrt_epsi:wN #1;
  {
    \__fp_fixed_sub:wwn {15000}{0000}{0000}{0000}{0000}{0000}; #1;
    \__fp_ep_isqrt_epsii:wwN #1;
    \__fp_ep_isqrt_epsii:wwN #1;
    \__fp_ep_isqrt_epsii:wwN #1;
  }
\cs_new:Npn \__fp_ep_isqrt_epsii:wwN #1; #2;
  {
    \__fp_fixed_mul:wwn #1; #1;
    \__fp_fixed_mul_sub_back:wwwn #2;
      {15000}{0000}{0000}{0000}{0000}{0000};
    \__fp_fixed_mul:wwn #1;
  }
```

(*End definition for* `\__fp_ep_isqrt_epsi:wN` *and* `\__fp_ep_isqrt_epsii:wwN`.)

29.11 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.

```
\cs_new:Npn \__fp_ep_to_float:wwN #1,
  { + \__int_eval:w #1 \__fp_fixed_to_float:wN }
\cs_new:Npn \__fp_ep_inv_to_float:wwN #1,#2;
  {
    \__fp_ep_div:wwwwn 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1,#2;
    \__fp_ep_to_float:wwN
  }
```

(*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.

```
\cs_new:Npn \__fp_fixed_inv_to_float:wN
  { \__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.

```
\cs_new:Npn \__fp_fixed_to_float_rad:wN #1;
  {
    \__fp_fixed_mul:wwn #1; {5729}{5779}{5130}{8232}{0876}{7981};
    { \__fp_ep_to_float:wwN 2, }
  }
```

(*End definition for* `\__fp_fixed_to_float_rad:wN`.)

`\__fp_fixed_to_float:wN`
`\__fp_fixed_to_float:Nw`

yields

⟨*exponent'*⟩ ; {⟨a'_1⟩} {⟨a'_2⟩} {⟨a'_3⟩} {⟨a'_4⟩} ;

And the `to_fixed` version gives six brace groups instead of 4, ensuring that $1000 \leq \langle a'_1\rangle \leq 9999$. 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.[8]

```
\cs_new:Npn \__fp_fixed_to_float:Nw #1#2; { \__fp_fixed_to_float:wN #2; #1 }
\cs_new:Npn \__fp_fixed_to_float:wN #1#2#3#4#5#6; #7
  {
    + \__int_eval:w \c_four % for the 8-digit-at-the-start thing.
    \exp_after:wN \exp_after:wN
    \exp_after:wN \__fp_fixed_to_loop:N
    \exp_after:wN \use_none:n
    \int_use:N \__int_eval:w
      1 0000 0000 + #1   \exp_after:wN \__fp_use_none_stop_f:n
      \__int_value:w   1#2 \exp_after:wN \__fp_use_none_stop_f:n
      \__int_value:w 1#3#4 \exp_after:wN \__fp_use_none_stop_f:n
      \__int_value:w 1#5#6
    \exp_after:wN ;
    \exp_after:wN ;
  }
\cs_new:Npn \__fp_fixed_to_loop:N #1
  {
    \if_meaning:w 0 #1
      - \c_one
      \exp_after:wN \__fp_fixed_to_loop:N
    \else:
      \exp_after:wN \__fp_fixed_to_loop_end:w
      \exp_after:wN #1
    \fi:
  }
\cs_new:Npn \__fp_fixed_to_loop_end:w #1 #2 ;
  {
    \if_meaning:w ; #1
      \exp_after:wN \__fp_fixed_to_float_zero:w
    \else:
      \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
      \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
      \exp_after:wN \__fp_fixed_to_float_pack:ww
      \exp_after:wN ;
    \fi:
    #1 #2 0000 0000 0000 0000 ;
  }
\cs_new:Npn \__fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
  {
```

[8] Bruno: I must double check this assumption.

```
    - \c_two * \c__fp_max_exponent_int ;
    {0000} {0000} {0000} {0000} ;
  }
\cs_new:Npn \__fp_fixed_to_float_pack:ww #1 ; #2#3 ; ;
  {
    \if_int_compare:w #2 > \c_four
      \exp_after:wN \__fp_fixed_to_float_round_up:wnnnnw
    \fi:
    ; #1 ;
  }
\cs_new:Npn \__fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
  {
    \exp_after:wN \__fp_basics_pack_high:NNNNNw
    \int_use:N \__int_eval:w 1 #1#2
      \exp_after:wN \__fp_basics_pack_low:NNNNNw
      \int_use:N \__int_eval:w 1 #3#4 + \c_one ;
  }
```

(*End definition for* `\__fp_fixed_to_float:wN` *and* `\__fp_fixed_to_float:Nw`*.*)

⟨/initex | package⟩

30 l3fp-expo implementation

⟨*initex | package⟩

⟨@@=fp⟩

30.1 Logarithm

30.1.1 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?

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 $5 \le c < 50$ such that $0.91 \le ac/5 < 1.1$, and use the relation

$$\ln(a \cdot 10^b) = b \cdot \ln(10) - \ln(c/5) + \ln(ac/5).$$

The logarithms $\ln(10)$ and $\ln(c/5)$ are looked up in a table. The last term is computed using the following Taylor series of ln near 1:

$$\ln\left(\frac{ac}{5}\right) = \ln\left(\frac{1+t}{1-t}\right) = 2t\left(1 + t^2\left(\frac{1}{3} + t^2\left(\frac{1}{5} + t^2\left(\frac{1}{7} + t^2\left(\frac{1}{9} + \cdots\right)\right)\right)\right)\right)$$

where $t = 1 - 10/(ac + 5)$. We can now see one reason for the choice of $ac \sim 5$: then $ac + 5 = 10(1 - \epsilon)$ with $-0.05 < \epsilon \leq 0.045$, hence

$$t = \frac{\epsilon}{1-\epsilon} = \epsilon(1+\epsilon)(1+\epsilon^2)(1+\epsilon^4)\ldots,$$

is not too difficult to compute.

30.1.2 Some constants

\c__fp_ln_i_fixed_tl
\c__fp_ln_ii_fixed_tl
\c__fp_ln_iii_fixed_tl
\c__fp_ln_iv_fixed_tl
\c__fp_ln_vi_fixed_tl
\c__fp_ln_vii_fixed_tl
\c__fp_ln_viii_fixed_tl
\c__fp_ln_ix_fixed_tl
\c__fp_ln_x_fixed_tl

A few values of the logarithm as extended fixed point numbers. Those are needed in the implementation. It turns out that we don't need the value of ln(5).

```
\tl_const:Nn \c__fp_ln_i_fixed_tl   { {0000}{0000}{0000}{0000}{0000}{0000} }
\tl_const:Nn \c__fp_ln_ii_fixed_tl  { {6931}{4718}{0559}{9453}{0941}{7232} }
\tl_const:Nn \c__fp_ln_iii_fixed_tl {{10986}{1228}{8668}{1096}{9139}{5245} }
\tl_const:Nn \c__fp_ln_iv_fixed_tl  {{13862}{9436}{1119}{8906}{1883}{4464} }
\tl_const:Nn \c__fp_ln_vi_fixed_tl  {{17917}{5946}{9228}{0550}{0081}{2477} }
\tl_const:Nn \c__fp_ln_vii_fixed_tl {{19459}{1014}{9055}{3133}{0510}{5353} }
\tl_const:Nn \c__fp_ln_viii_fixed_tl{{20794}{4154}{1679}{8359}{2825}{1696} }
\tl_const:Nn \c__fp_ln_ix_fixed_tl  {{21972}{2457}{7336}{2193}{8279}{0490} }
\tl_const:Nn \c__fp_ln_x_fixed_tl   {{23025}{8509}{2994}{0456}{8401}{7991} }
```

(*End definition for* `\c__fp_ln_i_fixed_tl` *and others.*)

30.1.3 Sign, exponent, and special numbers

__fp_ln_o:w

The logarithm of negative numbers (including $-\infty$ and -0) raises the "invalid" exception. The logarithm of $+0$ is $-\infty$, raising a division by zero exception. The logarithm of $+\infty$ or a `nan` is itself. Positive normal numbers call `\__fp_ln_npos_o:w`.

```
\cs_new:Npn \__fp_ln_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_meaning:w 2 #3
      \__fp_case_use:nw { \__fp_invalid_operation_o:nw { ln } }
    \fi:
    \if_case:w #2 \exp_stop_f:
      \__fp_case_use:nw
        { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
    \or:
    \else:
      \__fp_case_return_same_o:w
    \fi:
    \__fp_ln_npos_o:w \s__fp \__fp_chk:w #2#3#4;
  }
```

(*End definition for* `\__fp_ln_o:w`.)

30.1.4 Absolute ln

__fp_ln_npos_o:w

We catch the case of a significand very close to 0.1 or to 1. In all other cases, the final result is at least 10^{-4}, and then an error of $0.5 \cdot 10^{-20}$ is acceptable.

```
\cs_new:Npn \__fp_ln_npos_o:w \s__fp \__fp_chk:w 10#1#2#3;
```

```
  { %^^A todo: ln(1) should be "exact zero", not "underflow"
    \exp_after:wN \__fp_sanitize:Nw
    \__int_value:w % for the overall sign
      \if_int_compare:w #1 < \c_one
        2
      \else:
        0
      \fi:
      \exp_after:wN \exp_stop_f:
      \int_use:N \__int_eval:w % for the exponent
        \__fp_ln_significand:NNNNnnnN #2#3
        \__fp_ln_exponent:wn {#1}
  }
```

(*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.

```
\cs_new:Npn \__fp_ln_significand:NNNNnnnN #1#2#3#4
  {
    \exp_after:wN \__fp_ln_x_ii:wnnnn
    \__int_value:w
      \if_case:w #1 \exp_stop_f:
      \or:
        \if_int_compare:w #2 < \c_four
          \__int_eval:w \c_ten - #2
        \else:
          6
        \fi:
      \or: 4
      \or: 3
      \or: 2
      \or: 2
      \or: 2
      \else: 1
      \fi:
    ; { #1 #2 #3 #4 }
  }
```

(*End definition for* `\__fp_ln_significand:NNNNnnnN`.)

__fp_ln_x_ii:wnnnn

We have thus found c. It is chosen such that $0.7 \leq ac < 1.4$ in all cases. Compute $1 + x = 1 + ac \in [1.7, 2.4)$.

```
\cs_new:Npn \__fp_ln_x_ii:wnnnn #1; #2#3#4#5
  {
    \exp_after:wN \__fp_ln_div_after:Nw
    \cs:w c__fp_ln_ \tex_romannumeral:D #1 _fixed_tl \exp_after:wN \cs_end:
```

```
    \__int_value:w
      \exp_after:wN \__fp_ln_x_iv:wnnnnnnnn
      \int_use:N \__int_eval:w
        \exp_after:wN \__fp_ln_x_iii_var:NNNNNw
        \int_use:N \__int_eval:w 9999 9999 + #1*#2#3 +
          \exp_after:wN \__fp_ln_x_iii:NNNNNw
          \int_use:N \__int_eval:w 1 0000 0000 + #1*#4#5 ;
    {20000} {0000} {0000} {0000}
  } %^^A todo: reoptimize (a generalization attempt failed).
\cs_new:Npn \__fp_ln_x_iii:NNNNNw #1 #2#3#4#5 #6; { #1; {#2#3#4#5} {#6} }
\cs_new:Npn \__fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6;
  {
    #1#2#3#4#5 + \c_one ;
    {#1#2#3#4#5} {#6}
  }
```

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 $1.7 \cdot 10^4 \le y \le 2.4 \cdot 10^4$, and convexity, we get

$$\begin{aligned}
10^4 A &= 2 \cdot 10^4 \\
10^4 B &\le 10^8 \frac{A}{y} + 1.6y \le 4.7 \cdot 10^4 \\
10^4 C &\le 10^8 \frac{B}{y} + 1.6y \le 5.8 \cdot 10^4 \\
10^4 D &\le 10^8 \frac{C}{y} + 1.6y \le 6.3 \cdot 10^4 \\
10^4 E &\le 10^8 \frac{D}{y} + 1.6y \le 6.5 \cdot 10^4 \\
10^4 F &\le 10^8 \frac{E}{y} + 1.6y \le 6.6 \cdot 10^4
\end{aligned}$$

Note that we compute more steps than for division: since t is not the end result, we need to know it with more accuracy (on the other hand, the ending is much simpler, as we don't need an exact rounding for transcendental functions, but just a faithful rounding).[9]

`\__fp_ln_x_iv:wnnnnnnnn` ⟨*1 or 2*⟩ ⟨*8d*⟩ `;` `{`⟨*4d*⟩`}` `{`⟨*4d*⟩`}` ⟨*fixed-tl*⟩

The number is x. Compute y by adding 1 to the five first digits.

```
\cs_new:Npn \__fp_ln_x_iv:wnnnnnnnn #1; #2#3#4#5 #6#7#8#9
  {
    \exp_after:wN \__fp_div_significand_pack:NNN
    \int_use:N \__int_eval:w
    \__fp_ln_div_i:w #1 ;
      #6 #7 ; {#8} {#9}
      {#2} {#3} {#4} {#5}
      { \exp_after:wN \__fp_ln_div_ii:wwn \__int_value:w #1 }
      { \exp_after:wN \__fp_ln_div_ii:wwn \__int_value:w #1 }
      { \exp_after:wN \__fp_ln_div_ii:wwn \__int_value:w #1 }
      { \exp_after:wN \__fp_ln_div_ii:wwn \__int_value:w #1 }
      { \exp_after:wN \__fp_ln_div_vi:wwn \__int_value:w #1 }
  }
\cs_new:Npn \__fp_ln_div_i:w #1;
  {
    \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
    \int_use:N \__int_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
  }
\cs_new:Npn \__fp_ln_div_ii:wwn #1; #2;#3 % y; B1;B2 <- for k=1
  {
    \exp_after:wN \__fp_div_significand_pack:NNN
    \int_use:N \__int_eval:w
      \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
      \int_use:N \__int_eval:w 999999 + #2 #3 / #1 ; % Q2
```

[9]Bruno: to be completed.

```
    #2 #3 ;
  }
\cs_new:Npn \__fp_ln_div_vi:wwn #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
  {
    \exp_after:wN \__fp_div_significand_pack:NNN
    \int_use:N \__int_eval:w 1000000 + #2 #3 / #1 ; % Q6
  }
```

We now have essentially[10]

> __fp_ln_div_after:Nw ⟨*fixed tl*⟩ __fp_div_significand_pack:NNN $10^6 + Q_1$ __fp_div_significand_pack:NNN $10^6 + Q_2$ __fp_div_significand_pack:NNN $10^6 + Q_3$ __fp_div_significand_pack:NNN $10^6 + Q_4$ __fp_div_significand_pack:NNN $10^6 + Q_5$ __fp_div_significand_pack:NNN $10^6 + Q_6$; ⟨*exponent*⟩ ; ⟨*continuation*⟩

where ⟨*fixed tl*⟩ holds the logarithm of a number in $[1, 10]$, and ⟨*exponent*⟩ is the exponent. Also, the expansion is done backwards. Then __fp_div_significand_pack:NNN puts things in the correct order to add the Q_i together and put semicolons between each piece. Once those have been expanded, we get

> __fp_ln_div_after:Nw ⟨*fixed-tl*⟩ ⟨*1d*⟩ ; ⟨*4d*⟩ ; ⟨*4d*⟩ ; ⟨*4d*⟩ ; ⟨*4d*⟩ ; ⟨*4d*⟩ ; ⟨*4d*⟩ ; ⟨*exponent*⟩ ;

Just as with division, we know that the first two digits are 1 and 0 because of bounds on the final result of the division $2/(x+1)$, which is between roughly 0.8 and 1.2. We then compute $1 - 2/(x+1)$, after testing whether $2/(x+1)$ is greater than or smaller than 1.

```
\cs_new:Npn \__fp_ln_div_after:Nw #1#2;
  {
    \if_meaning:w 0 #2
      \exp_after:wN \__fp_ln_t_small:Nw
    \else:
      \exp_after:wN \__fp_ln_t_large:NNw
      \exp_after:wN -
    \fi:
    #1
  }
\cs_new:Npn \__fp_ln_t_small:Nw #1 #2; #3; #4; #5; #6; #7;
  {
    \exp_after:wN \__fp_ln_t_large:NNw
    \exp_after:wN + % <sign>
    \exp_after:wN #1
    \int_use:N \__int_eval:w 9999 - #2 \exp_after:wN ;
    \int_use:N \__int_eval:w 9999 - #3 \exp_after:wN ;
    \int_use:N \__int_eval:w 9999 - #4 \exp_after:wN ;
    \int_use:N \__int_eval:w 9999 - #5 \exp_after:wN ;
    \int_use:N \__int_eval:w 9999 - #6 \exp_after:wN ;
    \int_use:N \__int_eval:w 1 0000 - #7 ;
  }
```

[10] 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.

__fp_ln_t_large:NNw ⟨*sign*⟩⟨*fixed tl*⟩ ⟨t_1⟩; ⟨t_2⟩ ; ⟨t_3⟩; ⟨t_4⟩; ⟨t_5⟩ ; ⟨t_6⟩; ⟨*exponent*⟩ ; ⟨*continuation*⟩

Compute the square t^2, and keep t at the end with its sign. We know that t < 0.1765, so every piece has at most 4 digits. However, since we were not careful in __fp_ln_t_small:w, they can have less than 4 digits.

```
\cs_new:Npn \__fp_ln_t_large:NNw #1 #2 #3; #4; #5; #6; #7; #8;
  {
    \exp_after:wN \__fp_ln_square_t_after:w
    \int_use:N \__int_eval:w 9999 0000 + #3*#3
      \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
      \int_use:N \__int_eval:w 9999 0000 + 2*#3*#4
        \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
        \int_use:N \__int_eval:w 9999 0000 + 2*#3*#5 + #4*#4
          \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
          \int_use:N \__int_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
            \exp_after:wN \__fp_ln_square_t_pack:NNNNNw
            \int_use:N \__int_eval:w 1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
              + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
              % ; ; ;
    \exp_after:wN \__fp_ln_twice_t_after:w
    \int_use:N \__int_eval:w -1 + 2*#3
      \exp_after:wN \__fp_ln_twice_t_pack:Nw
      \int_use:N \__int_eval:w 9999 + 2*#4
        \exp_after:wN \__fp_ln_twice_t_pack:Nw
        \int_use:N \__int_eval:w 9999 + 2*#5
          \exp_after:wN \__fp_ln_twice_t_pack:Nw
          \int_use:N \__int_eval:w 9999 + 2*#6
            \exp_after:wN \__fp_ln_twice_t_pack:Nw
            \int_use:N \__int_eval:w 9999 + 2*#7
              \exp_after:wN \__fp_ln_twice_t_pack:Nw
              \int_use:N \__int_eval:w 10000 + 2*#8 ; ;
    { \__fp_ln_c:NwNw #1 }
    #2
  }
\cs_new:Npn \__fp_ln_twice_t_pack:Nw #1 #2; { + #1 ; {#2} }
\cs_new:Npn \__fp_ln_twice_t_after:w #1; { ;;; {#1} }
\cs_new:Npn \__fp_ln_square_t_pack:NNNNNw #1 #2#3#4#5 #6;
  { + #1#2#3#4#5 ; {#6} }
\cs_new:Npn \__fp_ln_square_t_after:w 1 0 #1#2#3 #4;
  { \__fp_ln_Taylor:wwNw {0#1#2#3} {#4} }
```

(*End 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;
```

[11]

This uses the routine for dividing a number by a small integer ($< 10^4$).

```
\cs_new:Npn \__fp_ln_Taylor:wwNw
  { \__fp_ln_Taylor_loop:www 21 ; {0000}{0000}{0000}{0000}{0000}{0000} ; }
\cs_new:Npn \__fp_ln_Taylor_loop:www #1; #2; #3;
  {
    \if_int_compare:w #1 = \c_one
      \__fp_ln_Taylor_break:w
    \fi:
    \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl ; #1;
    \__fp_fixed_add:wwn #2;
    \__fp_fixed_mul:wwn #3;
    {
      \exp_after:wN \__fp_ln_Taylor_loop:www
      \int_use:N \__int_eval:w #1 - \c_two ;
    }
    #3;
  }
\cs_new:Npn \__fp_ln_Taylor_break:w \fi: #1 \__fp_fixed_add:wwn #2#3; #4 ;;
  {
    \fi:
    \exp_after:wN \__fp_fixed_mul:wwn
    \exp_after:wN { \int_use:N \__int_eval:w 10000 + #2 } #3;
  }
```

(*End definition for* `\__fp_ln_Taylor:wwNw`. *This function is documented on page* **??**.)

`\__fp_ln_c:NwNw`

`\__fp_ln_c:NwNw` ⟨*sign*⟩ `{`⟨r_1⟩`}` `{`⟨r_2⟩`}` `{`⟨r_3⟩`}` `{`⟨r_4⟩`}` `{`⟨r_5⟩`}` `{`⟨r_6⟩`}` `;` ⟨*fixed tl*⟩ ⟨*exponent*⟩ `;` ⟨*continuation*⟩

We are now reduced to finding $\ln(c)$ and ⟨*exponent*⟩ $\ln(10)$ in a table, and adding it to the mixture. The first step is to get $\ln(c) - \ln(x) = -\ln(a)$, then we get `b` $\ln(10)$ and add or subtract.

For now, $\ln(x)$ is given as $\cdot 10^0$. Unless both the exponent is 1 and $c = 1$, we shift to working in units of $\cdot 10^4$, since the final result will be at least $\ln(10/7) \simeq 0.35$.[12]

[11] Bruno: add explanations.

[12] Bruno: that was wrong at some point, I must check.

```
\cs_new:Npn \__fp_ln_c:NwNw #1 #2; #3
  {
    \if_meaning:w + #1
      \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_sub:wwn
    \else:
      \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_add:wwn
    \fi:
    #3 ; #2 ;
  }
```

[13]

(*End definition for* `\__fp_ln_c:NwNw`. *This function is documented on page* **??**.)

`\__fp_ln_exponent:wn`

`\__fp_ln_exponent:wn` {⟨s_1⟩} {⟨s_2⟩} {⟨s_3⟩} {⟨s_4⟩} {⟨s_5⟩} {⟨s_6⟩} ; {⟨*exponent*⟩}

Compute ⟨*exponent*⟩ times ln(10). Apart from the cases where ⟨*exponent*⟩ is 0 or 1, the result will necessarily be at least $\ln(10) \simeq 2.3$ in magnitude. We can thus drop the least significant 4 digits. In the case of a very large (positive or negative) exponent, we can (and we need to) drop 4 additional digits, since the result is of order 10^4. Naively, one would think that in both cases we can drop 4 more digits than we do, but that would be slightly too tight for rounding to happen correctly. Besides, we already have addition and subtraction for 24 digits fixed point numbers.

```
\cs_new:Npn \__fp_ln_exponent:wn #1; #2
  {
    \if_case:w #2 \exp_stop_f:
      \c_zero \__fp_case_return:nw { \__fp_fixed_to_float:Nw 2 }
    \or:
      \exp_after:wN \__fp_ln_exponent_one:ww \__int_value:w
    \else:
      \if_int_compare:w #2 > \c_zero
        \exp_after:wN \__fp_ln_exponent_small:NNww
        \exp_after:wN 0
        \exp_after:wN \__fp_fixed_sub:wwn \__int_value:w
      \else:
        \exp_after:wN \__fp_ln_exponent_small:NNww
        \exp_after:wN 2
        \exp_after:wN \__fp_fixed_add:wwn \__int_value:w -
      \fi:
    \fi:
    #2; #1;
  }
```

Now we painfully write all the cases.[14] No overflow nor underflow can happen, except when computing `ln(1)`.

```
\cs_new:Npn \__fp_ln_exponent_one:ww 1; #1;
  {
    \c_zero
    \exp_after:wN \__fp_fixed_sub:wwn \c__fp_ln_x_fixed_tl ; #1;
```

[13]Bruno: this ***must*** be updated with correct values!
[14]Bruno: do rounding.

```
    \__fp_fixed_to_float:wN 0
  }
```

For small exponents, we just drop one block of digits, and set the exponent of the log to 4 (minus any shift coming from leading zeros in the conversion from fixed point to floating point). Note that here the exponent has been made positive.

```
\cs_new:Npn \__fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
  {
    \c_four
    \exp_after:wN \__fp_fixed_mul:wwn
      \c__fp_ln_x_fixed_tl ;
      {#3}{0000}{0000}{0000}{0000}{0000} ;
    #2
      {0000}{#4}{#5}{#6}{#7}{#8};
    \__fp_fixed_to_float:wN #1
  }
```

(*End definition for* `\__fp_ln_exponent:wn`. *This function is documented on page* **??**.)

30.2 Exponential

30.2.1 Sign, exponent, and special numbers

`\__fp_exp_o:w`

```
\cs_new:Npn \__fp_exp_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w #2 \exp_stop_f:
      \__fp_case_return_o:Nw \c_one_fp
    \or:
      \exp_after:wN \__fp_exp_normal:w
    \or:
      \if_meaning:w 0 #3
        \exp_after:wN \__fp_case_return_o:Nw
        \exp_after:wN \c_inf_fp
      \else:
        \exp_after:wN \__fp_case_return_o:Nw
        \exp_after:wN \c_zero_fp
      \fi:
    \or:
      \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2#3#4;
  }
```

(*End definition for* `\__fp_exp_o:w`.)

`\__fp_exp_normal:w`
`\__fp_exp_pos:Nnwnw`

```
\cs_new:Npn \__fp_exp_normal:w \s__fp \__fp_chk:w 1#1
  {
    \if_meaning:w 0 #1
      \__fp_exp_pos:NNwnw + \__fp_fixed_to_float:wN
```

```
    \else:
      \__fp_exp_pos:NNwnw - \__fp_fixed_inv_to_float:wN
    \fi:
  }
\cs_new:Npn \__fp_exp_pos:NNwnw #1#2#3 \fi: #4#5;
  {
    \fi:
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN 0
    \__int_value:w #1 \__int_eval:w
      \if_int_compare:w #4 < - \c_eight
        \c_one
        \exp_after:wN \__fp_add_big_i_o:wNww
        \int_use:N \__int_eval:w \c_one - #4 ;
        0 {1000}{0000}{0000}{0000} ; #5;
        \tex_romannumeral:D
      \else:
        \if_int_compare:w #4 > \c_five % cf \c__fp_max_exponent_int
          \exp_after:wN \__fp_exp_overflow:
          \tex_romannumeral:D
        \else:
          \if_int_compare:w #4 < \c_zero
            \exp_after:wN \use_i:nn
          \else:
            \exp_after:wN \use_ii:nn
          \fi:
          {
            \c_zero
            \__fp_decimate:nNnnnn { - #4 }
              \__fp_exp_Taylor:Nnnwn
          }
          {
            \__fp_decimate:nNnnnn { \c_sixteen - #4 }
              \__fp_exp_pos_large:NnnNwn
          }
          #5
          {#4}
          #1 #2 0
          \tex_romannumeral:D
        \fi:
      \fi:
    \exp_after:wN \c_zero
  }
\cs_new:Npn \__fp_exp_overflow:
  { + \c_two * \c__fp_max_exponent_int ; {1000} {0000} {0000} {0000} ; }
```

(*End definition for* `\__fp_exp_normal:w` *and* `\__fp_exp_pos:Nnwnw`.)

`\__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.

```
\cs_new:Npn \__fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
  {
    #6
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_pack_twice_four:wNNNNNNNN
    \__fp_exp_Taylor_ii:ww
    ; #2#3#4 0000 0000 ;
  }
\cs_new:Npn \__fp_exp_Taylor_ii:ww #1; #2;
  { \__fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s__stop }
\cs_new:Npn \__fp_exp_Taylor_loop:www #1; #2; #3;
  {
    \if_int_compare:w #1 = \c_one
      \exp_after:wN \__fp_exp_Taylor_break:Nww
    \fi:
    \__fp_fixed_div_int:wwN #3 ; #1 ;
    \__fp_fixed_add_one:wN
    \__fp_fixed_mul:wwn #2 ;
    {
      \exp_after:wN \__fp_exp_Taylor_loop:www
      \int_use:N \__int_eval:w #1 - 1 ;
      #2 ;
    }
  }
\cs_new:Npn \__fp_exp_Taylor_break:Nww #1 #2; #3 \s__stop
  { \__fp_fixed_add_one:wN #2 ; }
```

(*End definition for* `\__fp_exp_Taylor:Nnnwn`*.*)

__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

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.

```
\cs_new:Npn \__fp_exp_pos_large:NnnNwn #1#2#3 #4#5; #6
  {
    \exp_after:wN \exp_after:wN
    \cs:w __fp_exp_large_\tex_romannumeral:D #6:wN \exp_after:wN \cs_end:
    \exp_after:wN \c__fp_one_fixed_tl
    \exp_after:wN ;
```

```
    \__int_value:w #3 #4 \exp_stop_f:
    #5 00000 ;
  }
\cs_new:Npn \__fp_exp_large:w #1 \or: #2 \fi:
  { \fi: \__fp_fixed_mul:wwn #1; }
\cs_new:Npn \__fp_exp_large_v:wN #1; #2
  {
    \if_case:w #2 ~           \exp_after:wN \__fp_fixed_continue:wn  \or:
      +  4343 \__fp_exp_large:w {8806}{8182}{2566}{2921}{5872}{6150} \or:
      +  8686 \__fp_exp_large:w {7756}{0047}{2598}{6861}{0458}{3204} \or:
      + 13029 \__fp_exp_large:w {6830}{5723}{7791}{4884}{1932}{7351} \or:
      + 17372 \__fp_exp_large:w {6015}{5609}{3095}{3052}{3494}{7574} \or:
      + 21715 \__fp_exp_large:w {5297}{7951}{6443}{0315}{3251}{3576} \or:
      + 26058 \__fp_exp_large:w {4665}{6719}{0099}{3379}{5527}{2929} \or:
      + 30401 \__fp_exp_large:w {4108}{9724}{3326}{3186}{5271}{5665} \or:
      + 34744 \__fp_exp_large:w {3618}{6973}{3140}{0875}{3856}{4102} \or:
      + 39087 \__fp_exp_large:w {3186}{9209}{6113}{3900}{6705}{9685} \or:
    \fi:
    #1;
    \__fp_exp_large_iv:wN
  }
\cs_new:Npn \__fp_exp_large_iv:wN #1; #2
  {
    \if_case:w #2 ~          \exp_after:wN \__fp_fixed_continue:wn  \or:
      +  435 \__fp_exp_large:w {1970}{0711}{1401}{7046}{9938}{8888} \or:
      +  869 \__fp_exp_large:w {3881}{1801}{9428}{4368}{5764}{8232} \or:
      + 1303 \__fp_exp_large:w {7646}{2009}{8905}{4704}{8893}{1073} \or:
      + 1738 \__fp_exp_large:w {1506}{3559}{7005}{0524}{9009}{7592} \or:
      + 2172 \__fp_exp_large:w {2967}{6283}{8402}{3667}{0689}{6630} \or:
      + 2606 \__fp_exp_large:w {5846}{4389}{5650}{2114}{7278}{5046} \or:
      + 3041 \__fp_exp_large:w {1151}{7900}{5080}{6878}{2914}{4154} \or:
      + 3475 \__fp_exp_large:w {2269}{1083}{0850}{6857}{8724}{4002} \or:
      + 3909 \__fp_exp_large:w {4470}{3047}{3316}{5442}{6408}{6591} \or:
    \fi:
    #1;
    \__fp_exp_large_iii:wN
  }
\cs_new:Npn \__fp_exp_large_iii:wN #1; #2
  {
    \if_case:w #2 ~         \exp_after:wN \__fp_fixed_continue:wn  \or:
      +  44 \__fp_exp_large:w {2688}{1171}{4181}{6135}{4484}{1263} \or:
      +  87 \__fp_exp_large:w {7225}{9737}{6812}{5749}{2581}{7748} \or:
      + 131 \__fp_exp_large:w {1942}{4263}{9524}{1255}{9365}{8421} \or:
      + 174 \__fp_exp_large:w {5221}{4696}{8976}{4143}{9505}{8876} \or:
      + 218 \__fp_exp_large:w {1403}{5922}{1785}{2837}{4107}{3977} \or:
      + 261 \__fp_exp_large:w {3773}{0203}{0092}{9939}{8234}{0143} \or:
      + 305 \__fp_exp_large:w {1014}{2320}{5473}{5004}{5094}{5533} \or:
      + 348 \__fp_exp_large:w {2726}{3745}{7211}{2566}{5673}{6478} \or:
      + 391 \__fp_exp_large:w {7328}{8142}{2230}{7421}{7051}{8866} \or:
    \fi:
```

```
    #1;
    \__fp_exp_large_ii:wN
  }
\cs_new:Npn \__fp_exp_large_ii:wN #1; #2
  {
    \if_case:w #2 ~        \exp_after:wN \__fp_fixed_continue:wn  \or:
      +  5 \__fp_exp_large:w {2202}{6465}{7948}{0671}{6516}{9579} \or:
      +  9 \__fp_exp_large:w {4851}{6519}{5409}{7902}{7796}{9107} \or:
      + 14 \__fp_exp_large:w {1068}{6474}{5815}{2446}{2146}{9905} \or:
      + 18 \__fp_exp_large:w {2353}{8526}{6837}{0199}{8540}{7900} \or:
      + 22 \__fp_exp_large:w {5184}{7055}{2858}{7072}{4640}{8745} \or:
      + 27 \__fp_exp_large:w {1142}{0073}{8981}{5684}{2836}{6296} \or:
      + 31 \__fp_exp_large:w {2515}{4386}{7091}{9167}{0062}{6578} \or:
      + 35 \__fp_exp_large:w {5540}{6223}{8439}{3510}{0525}{7117} \or:
      + 40 \__fp_exp_large:w {1220}{4032}{9431}{7840}{8020}{0271} \or:
    \fi:
    #1;
    \__fp_exp_large_i:wN
  }
\cs_new:Npn \__fp_exp_large_i:wN #1; #2
  {
    \if_case:w #2 ~       \exp_after:wN \__fp_fixed_continue:wn  \or:
      + 1 \__fp_exp_large:w {2718}{2818}{2845}{9045}{2353}{6029} \or:
      + 1 \__fp_exp_large:w {7389}{0560}{9893}{0650}{2272}{3043} \or:
      + 2 \__fp_exp_large:w {2008}{5536}{9231}{8766}{7740}{9285} \or:
      + 2 \__fp_exp_large:w {5459}{8150}{0331}{4423}{9078}{1103} \or:
      + 3 \__fp_exp_large:w {1484}{1315}{9102}{5766}{0342}{1116} \or:
      + 3 \__fp_exp_large:w {4034}{2879}{3492}{7351}{2260}{8387} \or:
      + 4 \__fp_exp_large:w {1096}{6331}{5842}{8458}{5992}{6372} \or:
      + 4 \__fp_exp_large:w {2980}{9579}{8704}{1728}{2747}{4359} \or:
      + 4 \__fp_exp_large:w {8103}{0839}{2757}{5384}{0077}{1000} \or:
    \fi:
    #1;
    \__fp_exp_large_:wN
  }
\cs_new:Npn \__fp_exp_large_:wN #1; #2
  {
    \if_case:w #2 ~       \exp_after:wN \__fp_fixed_continue:wn  \or:
      + 1 \__fp_exp_large:w {1105}{1709}{1807}{5647}{6248}{1171} \or:
      + 1 \__fp_exp_large:w {1221}{4027}{5816}{0169}{8339}{2107} \or:
      + 1 \__fp_exp_large:w {1349}{8588}{0757}{6003}{1039}{8374} \or:
      + 1 \__fp_exp_large:w {1491}{8246}{9764}{1270}{3178}{2485} \or:
      + 1 \__fp_exp_large:w {1648}{7212}{7070}{0128}{1468}{4865} \or:
      + 1 \__fp_exp_large:w {1822}{1188}{0039}{0508}{9748}{7537} \or:
      + 1 \__fp_exp_large:w {2013}{7527}{0747}{0476}{5216}{2455} \or:
      + 1 \__fp_exp_large:w {2225}{5409}{2849}{2467}{6045}{7954} \or:
      + 1 \__fp_exp_large:w {2459}{6031}{1115}{6949}{6638}{0013} \or:
    \fi:
    #1;
    \__fp_exp_large_after:wwn
```

```
  }
\cs_new:Npn \__fp_exp_large_after:wwn #1; #2; #3
  {
    \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2; {} #3
    \__fp_fixed_mul:wwn #1;
  }
```

(*End definition for* `\__fp_exp_pos_large:NnnNwn` *and others.*)

30.3 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 $\texttt{nan}^0 = 1^{\texttt{nan}} = 1$, 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 a^b (`\__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.

```
\cs_new:cpn { __fp_ \iow_char:N \^ _o:ww }
    \s__fp \__fp_chk:w #1#2#3; \s__fp \__fp_chk:w #4#5#6;
  {
    \if_meaning:w 0 #4
      \__fp_case_return_o:Nw \c_one_fp
    \fi:
```

```
      \if_case:w #2 \exp_stop_f:
        \exp_after:wN \use_i:nn
      \or:
        \__fp_case_return_o:Nw \c_nan_fp
      \else:
        \exp_after:wN \__fp_pow_neg:www
        \tex_romannumeral:D -`0 \exp_after:wN \use:nn
      \fi:
      {
        \if_meaning:w 1 #1
          \exp_after:wN \__fp_pow_normal:ww
        \else:
          \exp_after:wN \__fp_pow_zero_or_inf:ww
        \fi:
        \s__fp \__fp_chk:w #1#2#3;
      }
      { \s__fp \__fp_chk:w #4#5#6; \s__fp \__fp_chk:w #1#2#3; }
      \s__fp \__fp_chk:w #4#5#6;
    }
```

(*End definition for* `\__fp_^_o:ww`.)

`\__fp_pow_zero_or_inf:ww` Raising -0 or $-\infty$ to `nan` yields `nan`. For other powers, the result is $+0$ if 0 is raised to a positive power or ∞ to a negative power, and $+\infty$ otherwise. Thus, if the type of a and the sign of b coincide, the result is 0, since those conveniently take the same possible values, 0 and 2. Otherwise, either $a = \pm 0$ 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.

```
  \cs_new:Npn \__fp_pow_zero_or_inf:ww \s__fp \__fp_chk:w #1#2; \s__fp \__fp_chk:w #3#4
    {
      \if_meaning:w 1 #4
        \__fp_case_return_same_o:w
      \fi:
      \if_meaning:w #1 #4
        \__fp_case_return_o:Nw \c_zero_fp
      \fi:
      \if_meaning:w 0 #1
        \__fp_case_use:nw
          {
            \__fp_division_by_zero_o:NNww \c_inf_fp ^
              \s__fp \__fp_chk:w #1 #2 ;
          }
      \else:
        \__fp_case_return_o:Nw \c_inf_fp
      \fi:
      \s__fp \__fp_chk:w #3#4
    }
```

(*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.

```
\cs_new:Npn \__fp_pow_normal:ww \s__fp \__fp_chk:w 1 #1#2#3; \s__fp \__fp_chk:w #4#5
  {
    \if_int_compare:w \pdftex_strcmp:D { #2 #3 }
              { 1 {1000} {0000} {0000} {0000} } = \c_zero
      \if_int_compare:w #4 #1 = 32 \exp_stop_f:
        \exp_after:wN \__fp_case_return_ii_o:ww
      \fi:
      \__fp_case_return_o:Nww \c_one_fp
    \fi:
    \if_case:w #4 \exp_stop_f:
    \or:
      \exp_after:wN \__fp_pow_npos:Nww
      \exp_after:wN #5
    \or:
      \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
      \if_int_compare:w #2 > \c_zero
        \exp_after:wN \__fp_case_return_o:Nww
        \exp_after:wN \c_inf_fp
      \else:
        \exp_after:wN \__fp_case_return_o:Nww
        \exp_after:wN \c_zero_fp
      \fi:
    \or:
      \__fp_case_return_ii_o:ww
    \fi:
    \s__fp \__fp_chk:w 1 #1 {#2} #3 ;
    \s__fp \__fp_chk:w #4 #5
  }
```

(*End 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 the exponential accurately, we need to know the digits of z up to the 16-th position. Since the exponential of 10^5 is infinite, we only need at most 21 digits, hence the fixed point result of `\__fp_ln_o:w` is precise enough for our needs. Start an integer expression for the decimal exponent of $e^{|z|}$. If z is negative, negate that decimal exponent, and prepare to take the inverse when converting from the fixed point to the floating point result.

```
\cs_new:Npn \__fp_pow_npos:Nww #1 \s__fp \__fp_chk:w 1#2#3
```

```
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN 0
    \__int_value:w
      \if:w #1 \if_int_compare:w #3 > \c_zero 0 \else: 2 \fi:
        \exp_after:wN \__fp_pow_npos_aux:NNnww
        \exp_after:wN +
        \exp_after:wN \__fp_fixed_to_float:wN
      \else:
        \exp_after:wN \__fp_pow_npos_aux:NNnww
        \exp_after:wN -
        \exp_after:wN \__fp_fixed_inv_to_float:wN
      \fi:
      {#3}
  }
```

(*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 $-\ln(x)$.

```
\cs_new:Npn \__fp_pow_npos_aux:NNnww #1#2#3#4#5; \s__fp \__fp_chk:w 1#6#7#8;
  {
    #1
    \__int_eval:w
      \__fp_ln_significand:NNNNnnnN #4#5
      \__fp_pow_exponent:wnN {#3}
      \__fp_fixed_mul:wwn #8 {0000}{0000} ;
      \__fp_pow_B:wwN #7;
      #1 #2 0 % fixed_to_float:wN
  }
\cs_new:Npn \__fp_pow_exponent:wnN #1; #2
  {
    \if_int_compare:w #2 > \c_zero
      \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % n\ln(10) - (-\ln(x))
      \exp_after:wN +
    \else:
      \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % -( |n|\ln(10) + (-\ln(x)) )
      \exp_after:wN -
    \fi:
    #2; #1;
  }
\cs_new:Npn \__fp_pow_exponent:Nwnnnnnw #1#2; #3#4#5#6#7#8;
  { %^^A todo: use that in ln.
    \exp_after:wN \__fp_fixed_mul_after:wwn
    \int_use:N \__int_eval:w \c__fp_leading_shift_int
      \exp_after:wN \__fp_pack:NNNNNw
      \int_use:N \__int_eval:w \c__fp_middle_shift_int
        #1#2*23025 - #1 #3
        \exp_after:wN \__fp_pack:NNNNNw
        \int_use:N \__int_eval:w \c__fp_middle_shift_int
```

```
          #1 #2*8509 - #1 #4
          \exp_after:wN \__fp_pack:NNNNNw
          \int_use:N \__int_eval:w \c__fp_middle_shift_int
            #1 #2*2994 - #1 #5
            \exp_after:wN \__fp_pack:NNNNNw
            \int_use:N \__int_eval:w \c__fp_middle_shift_int
              #1 #2*0456 - #1 #6
              \exp_after:wN \__fp_pack:NNNNNw
              \int_use:N \__int_eval:w \c__fp_trailing_shift_int
                #1 #2*8401 - #1 #7
                #1 ( #2*7991 - #8 ) / 1 0000 ; ;
  }
\cs_new:Npn \__fp_pow_B:wwN #1#2#3#4#5#6; #7;
  {
    \if_int_compare:w #7 < \c_zero
      \exp_after:wN \__fp_pow_C_neg:w \__int_value:w -
    \else:
      \if_int_compare:w #7 < 22 \exp_stop_f:
        \exp_after:wN \__fp_pow_C_pos:w \__int_value:w
      \else:
        \exp_after:wN \__fp_pow_C_overflow:w \__int_value:w
      \fi:
    \fi:
    #7 \exp_after:wN ;
    \int_use:N \__int_eval:w 10 0000 + #1 \__int_eval_end:
    #2#3#4#5#6 0000 0000 0000 0000 0000 0000 ; %^^A todo: how many 0?
  }
\cs_new:Npn \__fp_pow_C_overflow:w #1; #2; #3
  {
    + \c_two * \c__fp_max_exponent_int
    \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl ;
  }
\cs_new:Npn \__fp_pow_C_neg:w #1 ; 1
  {
    \exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w
    \prg_replicate:nn {#1} {0}
  }
\cs_new:Npn \__fp_pow_C_pos:w #1; 1
  { \__fp_pow_C_pos_loop:wN #1; }
\cs_new:Npn \__fp_pow_C_pos_loop:wN #1; #2
  {
    \if_meaning:w 0 #1
      \exp_after:wN \__fp_pow_C_pack:w
      \exp_after:wN #2
    \else:
      \if_meaning:w 0 #2
        \exp_after:wN \__fp_pow_C_pos_loop:wN \__int_value:w
      \else:
        \exp_after:wN \__fp_pow_C_overflow:w \__int_value:w
      \fi:
```

```
      \__int_eval:w #1 - \c_one \exp_after:wN ;
    \fi:
  }
\cs_new:Npn \__fp_pow_C_pack:w
  { \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), $a^b = \mathtt{a}^b$. If b is an odd integer (case 0), $a^b = -\mathtt{a}^b$, obtained by a call to `\__fp_pow_neg_aux:wNN`. Otherwise, the sign is undefined. This is invalid, unless $\mathtt{a}^b$ turns out to be $+0$ or `nan`, in which case we return that as a^b. In particular, since the underflow detection occurs before `\__fp_pow_neg:www` is called, `(-0.1)**(12345.6)` will give $+0$ rather than complaining that the sign is not defined.

```
\cs_new:Npn \__fp_pow_neg:www \s__fp \__fp_chk:w #1#2; #3; #4;
  {
    \if_case:w \__fp_pow_neg_case:w #4 ;
      \exp_after:wN \__fp_pow_neg_aux:wNN
    \or:
      \if_int_compare:w \__int_eval:w #1 / \c_two = \c_one
        \__fp_invalid_operation_o:Nww ^ #3; #4;
        \tex_romannumeral:D -`0
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_use_none_until_s:w
      \fi:
    \fi:
    \__fp_exp_after_o:w
    \s__fp \__fp_chk:w #1#2;
  }
\cs_new:Npn \__fp_pow_neg_aux:wNN #1 \s__fp \__fp_chk:w #2#3
  {
    \exp_after:wN \__fp_exp_after_o:w
    \exp_after:wN \s__fp
    \exp_after:wN \__fp_chk:w
    \exp_after:wN #2
    \int_use:N \__int_eval:w \c_two - #3 \__int_eval_end:
  }
```

(*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 `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 ⟨*digit*⟩ ⟨*digits*⟩ `;` which would be the digits surrounding the decimal period. If the ⟨*digits*⟩ are non-zero, the number is not an integer. Otherwise, check the parity of the ⟨*digit*⟩ and return `\c_zero` or `\c_minus_one`.

```
\cs_new:Npn \__fp_pow_neg_case:w \s__fp \__fp_chk:w #1#2#3;
  {
    \if_case:w #1 \exp_stop_f:
           \c_minus_one
    \or:   \__fp_pow_neg_case_aux:nnnnn #3
    \else: \c_one
    \fi:
  }
\cs_new:Npn \__fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
  {
    \if_int_compare:w #1 > \c_eight
      \if_int_compare:w #1 > \c_sixteen
        \c_minus_one
      \else:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_pow_neg_case_aux:NNNNNNNNw
        \prg_replicate:nn { \c_sixteen - #1 } { 0 } #4#5 ;
      \fi:
    \else:
      \if_int_compare:w #1 > \c_zero
        \if_int_compare:w #4#5 = \c_zero
          \exp_after:wN \exp_after:wN
          \exp_after:wN \__fp_pow_neg_case_aux:NNNNNNNNw
          \prg_replicate:nn { \c_eight - #1 } { 0 } #2#3 ;
        \else:
          \c_one
        \fi:
      \else:
        \c_one
      \fi:
    \fi:
  }
\cs_new:Npn \__fp_pow_neg_case_aux:NNNNNNNNw #1#2#3#4#5#6#7#8#9;
  {
    \if_int_compare:w 0 #9 = \c_zero
      \if_int_odd:w #8 \exp_stop_f:
        \c_zero
      \else:
        \c_minus_one
      \fi:
    \else:
      \c_one
    \fi:
  }
```

(*End definition for* `\__fp_pow_neg_case:w`, `\__fp_pow_neg_case_aux:nnnnn`, *and* `\__fp_pow_neg_case_aux:NNNNNNNNw`.)

⟨/initex | package⟩

31 l3fp-trig Implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

31.1 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.

31.1.1 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 $\sin(x) = \#3 \sin|x|$.

```
\cs_new:Npn \__fp_sin_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_case_return_same_o:w
    \or:   \__fp_case_use:nw
             {
               \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
               \__fp_ep_to_float:wwN #3 \c_zero
             }
    \or:   \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { sin } { sind } } }
    \else: \__fp_case_return_same_o:w
    \fi:
```

```
    \s__fp \__fp_chk:w #2 #3 #4;
  }
```

(*End definition for* `\__fp_sin_o:w`.)

`\__fp_cos_o:w` The cosine of ±0 is 1. The cosine of ±∞ 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 $\cos(x) = +\sin(\pi/2 + |x|)$.

```
\cs_new:Npn \__fp_cos_o:w #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_case_return_o:Nw \c_one_fp
    \or:   \__fp_case_use:nw
             {
               \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
               \__fp_ep_to_float:wwN 0 \c_two
             }
    \or:   \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3;
  }
```

(*End definition for* `\__fp_cos_o:w`.)

`\__fp_csc_o:w` The cosecant of ±0 is ±∞ 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 ±∞ raises an invalid operation exception. The cosecant of `nan` is itself. Otherwise, the `trig` function performs the argument reduction, and converts if necessary to radians before calling the same series as for sine, using the sign `#3`, a starting octant of 0, and inverting during the conversion from the fixed point sine to the floating point result, because $\csc(x) = \#3\big(\sin|x|\big)^{-1}$.

```
\cs_new:Npn \__fp_csc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
    \or:   \__fp_case_use:nw
             {
               \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
               \__fp_ep_inv_to_float:wwN #3 \c_zero
             }
    \or:   \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { csc } { cscd } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3 #4;
  }
```

(*End definition for* `\__fp_csc_o:w`.)

`\__fp_sec_o:w` The secant of ± 0 is 1. The secant of $\pm\infty$ raises an invalid operation exception. The secant of `nan` is itself. Otherwise, the `trig` function reduces the argument and turns it to radians before calling the same series as for sine, using a positive sign 0, a starting octant of 2, and inverting upon conversion, because $\sec(x) = +1/\sin(\pi/2 + |x|)$.

```
\cs_new:Npn \__fp_sec_o:w #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_case_return_o:Nw \c_one_fp
    \or:   \__fp_case_use:nw
             {
               \__fp_trig:NNNNNwn #1 \__fp_sin_series_o:NNwwww
               \__fp_ep_inv_to_float:wwN 0 \c_two
             }
    \or:   \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { sec } { secd } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3;
  }
```

(*End definition for* `\__fp_sec_o:w`.)

`\__fp_tan_o:w` The tangent of ± 0 or `nan` is the same floating point number. The tangent of $\pm\infty$ raises an invalid operation exception. Once more, the `trig` function does the argument reduction step and conversion to radians before calling `\__fp_tan_series_o:NNwwww`, with a sign `#3` and an initial octant of 1 (this shift is somewhat arbitrary). See `\__fp_cot_o:w` for an explanation of the 0 argument.

```
\cs_new:Npn \__fp_tan_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_case_return_same_o:w
    \or:   \__fp_case_use:nw
             {
               \__fp_trig:NNNNNwn #1
                 \__fp_tan_series_o:NNwwww 0 #3 \c_one
             }
    \or:   \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { tan } { tand } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3 #4;
  }
```

(*End definition for* `\__fp_tan_o:w`.)

`\__fp_cot_o:w`
`\__fp_cot_zero_o:Nfw`

The cotangent of ± 0 is $\pm\infty$ with the same sign, with a division by zero exception (see `\__fp_cot_zero_o:Nfw`. The cotangent of $\pm\infty$ raises an invalid operation exception. The cotangent of `nan` is itself. We use $\cot x = -\tan(\pi/2 + x)$, and the initial octant for the tangent was chosen to be 1, so the octant here starts at 3. The change in sign is obtained by feeding `\__fp_tan_series_o:NNwwww` two signs rather than just the sign of the argument: the first of those indicates whether we compute tangent or cotangent. Those signs are eventually combined.

```
\cs_new:Npn \__fp_cot_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w #2 \exp_stop_f:
           \__fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
    \or:   \__fp_case_use:nw
             {
               \__fp_trig:NNNNNwn #1
                 \__fp_tan_series_o:NNwwww 2 #3 \c_three
             }
    \or:   \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { cot } { cotd } } }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3 #4;
  }
\cs_new:Npn \__fp_cot_zero_o:Nfw #1#2#3 \fi:
  {
    \fi:
    \token_if_eq_meaning:NNTF 0 #1
      { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_inf_fp }
      { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_minus_inf_fp }
    {#2}
  }
```

(*End definition for* `\__fp_cot_o:w`.)

31.1.2 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.

```
\cs_new:Npn \__fp_trig:NNNNNwn #1#2#3#4#5 \s__fp \__fp_chk:w 1#6#7#8;
  {
    \exp_after:wN #2
    \exp_after:wN #3
    \exp_after:wN #4
    \int_use:N \__int_eval:w #5
      \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
      \if_int_compare:w #7 > #1 \c_zero \c_one
        #1 \__fp_trig_large:ww \__fp_trigd_large:ww
      \else:
        #1 \__fp_trig_small:ww \__fp_trigd_small:ww
      \fi:
    #7,#8{0000}{0000};
  }
```

(*End definition for* `\__fp_trig:NNNNNwn`.)

31.1.3 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.

```
\cs_new:Npn \__fp_trig_small:ww #1,#2;
  { \__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.

```
\cs_new:Npn \__fp_trigd_small:ww #1,#2;
  {
    \__fp_ep_mul_raw:wwwwN
      -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;
    \__fp_trig_small:ww
  }
```

(*End definition for* `\__fp_trigd_small:ww`.)

31.1.4 Argument reduction in degrees

`\__fp_trigd_large:ww`
`\__fp_trigd_large_auxi:nnnnwNNNN`
`\__fp_trigd_large_auxii:wNw`
`\__fp_trigd_large_auxiii:www`

Note that $25\times 360 = 9000$, so $10^{k+1} \equiv 10^k \pmod{360}$ for $k \geq 3$. When the exponent `#1` is very large, we can thus safely replace it by 22 (or even 19). We turn the floating point number into a fixed point number with two blocks of 8 digits followed by five blocks of 4 digits. The original float is $100 \times \langle block_1\rangle \cdots \langle block_3\rangle.\langle block_4\rangle \cdots \langle block_7\rangle$, or is equal to it modulo 360 if the exponent `#1` is very large. The first auxiliary finds $\langle block_1\rangle + \langle block_2\rangle \pmod 9$, 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 ⟨*block*$_4$⟩. 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 ⟨*blocks*⟩. Since the original exponent #1 is at least 2, those are all 0 and no precision is lost (#6 and #7 are four 0 each).

```
\cs_new:Npn \__fp_trigd_large:ww #1, #2#3#4#5#6#7;
  {
    \exp_after:wN \__fp_pack_eight:wNNNNNNNN
    \exp_after:wN \__fp_pack_eight:wNNNNNNNN
    \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
    \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
    \exp_after:wN \__fp_trigd_large_auxi:nnnnwNNNN
    \exp_after:wN ;
    \tex_romannumeral:D -`0
    \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
    #2#3#4#5#6#7 0000 0000 0000 !
  }
\cs_new:Npn \__fp_trigd_large_auxi:nnnnwNNNN #1#2#3#4#5; #6#7#8#9
  {
    \exp_after:wN \__fp_trigd_large_auxii:wNw
    \int_use:N \__int_eval:w #1 + #2
      - (#1 + #2 - \c_four) / \c_nine * \c_nine \__int_eval_end:
    #3;
    #4; #5{#6#7#8#9};
  }
\cs_new:Npn \__fp_trigd_large_auxii:wNw #1; #2#3;
  {
    + (#1#2 - \c_four) / \c_nine * \c_two
    \exp_after:wN \__fp_trigd_large_auxiii:www
    \int_use:N \__int_eval:w #1#2
      - (#1#2 - \c_four) / \c_nine * \c_nine \__int_eval_end: #3 ;
  }
\cs_new:Npn \__fp_trigd_large_auxiii:www #1; #2; #3!
  {
    \if_int_compare:w #1 < 4500 \exp_stop_f:
      \exp_after:wN \__fp_use_i_until_s:nw
      \exp_after:wN \__fp_fixed_continue:wn
    \else:
      + \c_one
    \fi:
    \__fp_fixed_sub:wwn {9000}{0000}{0000}{0000}{0000}{0000};
      {#1}#2{0000}{0000};
    { \__fp_trigd_small:ww 2, }
  }
```

(*End definition for* __fp_trigd_large:ww *and others.*)

31.1.5 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.

```
\cs_new_nopar:Npx \__fp_trig_inverse_two_pi:
  {
    \exp_not:n { \exp_after:wN \use_none:n \token_to_str:N }
    \cs:w , , !
    0000000000000000159154943091895335768883763372514362034459645740 ~
    456448747667344058896797634226535090113802766253085956072842727 ~
    5795803689291184611457865287796741073169983922923996693740907757 ~
    3077746396925307688717392896217397661693362390241723629011832380 ~
    1142226997557159404618900869026739561204894109369378440855287230 ~
    9994644340024867234773945961089832309678307490616698646280469944 ~
    8652187881574786566964241038995874139348609983868099199962442875 ~
    5851711788584311175187671605465475369880097394603647593337680593 ~
    0249449663530532715677550322032477781639716602294674811959816584 ~
    0606016803035998133911987498832786654435279755070016240677564388 ~
    8495713108801221993761476813777647378906330680464579784817613124 ~
    2731406996077502450029775985708905690279678513152521001631774602 ~
    0924811606240561456203146484089248459191435211575407556200871526 ~
    6068022171591407574745827225977462853998751553293908139817724093 ~
    5825479707332871904069997590765770784934703935898280871734256403 ~
    6689511662545705943327631268650026122717971153211259950438667945 ~
    0376255608363171169525975812822494162333431451061235368785631136 ~
    3669216714206974696012925057833605311960859450983955671870995474 ~
```

6510431623815517580839442979970999505254387566129445883306846050 ~
7852915151410404892988506388160776196993073410389995786918905980 ~
9373777206187543222718930136625526123878038753888110681406765434 ~
0828278526933426799556070790386060352738996245125995749276297023 ~
5940955843011648296411855777124057544494570217897697924094903272 ~
9477021664960356531815354400384068987471769158876319096650696440 ~
4776970687683656778104779795450353395758301881838687937766124814 ~
9530599655802190835987510351271290432315804987196868777594656634 ~
6221034204440855497850379273869429353661937782928735937843470323 ~
0237145837923557118636341929460183182291964165008783079331353497 ~
7909974586492902674506098936890945883050337030538054731232158094 ~
3197676032283131418980974982243833517435698984750103950068388003 ~
9786723599608024002739010874954854787923568261139948903268997427 ~
0834961149208289037767847430355045684560836714793084567233270354 ~
8539255620208683932409956221175331839402097079357077496549880868 ~
6066360968661967037474542102831219251846224834991161149566556037 ~
9696761399312829960776082779901007830360023382729879085402387615 ~
5744543092601191005433799838904654921248295160707285300522721023 ~
6017523313173179759311050328155109373913639645305792607180083617 ~
9548767246459804739772924481092009371257869183328958862839904358 ~
6866663975673445140950363732719174311388066383072592302759734506 ~
0548212778037065337783032170987734966568490800326988506741791464 ~
6835082816168533143361607309951498531198197337584442098416559541 ~
5225064339431286444038388356150879771645017064706751877456059160 ~
8716857857939226234756331711132998655941596890719850688744230057 ~
5191977056900382183925622033874235362568083541565172971088117217 ~
9593683256488518749974870855311659830610139214454460161488452770 ~
2511411070248521739745103866736403872860099674893173561812071174 ~
0478899368886556923078485023057057144063638632023685201074100574 ~
8592281115721968003978247595300166958522123034641877365043546764 ~
6456565971901123084767099309708591283646669191776938791433315566 ~
5066981321641521008957117286238426070678451760111345080069947684 ~
2235698962488051577598095339708085475059753626564903439445420581 ~
7886435683042000315095594743439252544850674914290864751442303321 ~
3324569511634945677539394240360905438335528292434220349484366151 ~
4663228602477666660495314065734357553014090827988091478669343492 ~
2737602634997829957018161964321233140475762897484082891174097478 ~
2637899181699939487497715198981872666294601830539583275209236350 ~
6853889228468247259972528300766856937583659722919824429747406163 ~
8183113958306744348516928597383237392662402434501997809940402189 ~
6134834273613676449913827154166063424829363741850612261086132119 ~
9863346284709941839942742955915628333990480382117501161211667205 ~
1912579303552929241134403116134112495318385926958490443846807849 ~
0973982808855297045153053991400988698840883654836652224668624087 ~
2540140400911787421220452307533473972538149403884190586842311594 ~
6322744339066125162393106283195323883392131534556381511752035108 ~
7459558201123754359768155340187407394340363397803881721004531691 ~
8295194879591767395417787924352761740724605939160273228287946819 ~
3649128949714953432552723591659298072479985806126900733218844526 ~
7943350455801952492566306204876616134365339920287545208555344144 ~

0990512982727454659118132223284051166615650709837557433729548631 ~
2041121716380915606161165732000083306114606181280326258695951602 ~
4632166138576614804719932707771316441201594960110632830520759583 ~
4850305079095584982982186740289838551383239570208076397550429225 ~
9847647071016426974384504309165864528360324933604354657237557916 ~
1366324120457809969715663402215880545794313282780055246132088901 ~
8742121092448910410052154968097113720754005710963406643135745439 ~
9159769435788920793425617783022237011486424925239248728713132021 ~
7667360756645598272609574156602343787436291321097485897150713073 ~
9104072643541417970572226547980381512759579124002534468048220261 ~
7342299001020483062463033796474678190501811830375153802879523433 ~
4195502135689770912905614317878792086205744999257897569018492103 ~
2420647138519113881475640209760554895793785141404145305151583964 ~
2823265406020603311891586570272086250269916393751527887360608114 ~
5569484210322407772727421651364234366992716340309405307480652685 ~
0930165892136921414312937134106157153714062039784761842650297807 ~
8606266969960809184223476335047746719017450451446166382846208240 ~
8673595102371302904443779408535034454426334130626307459513830310 ~
2293146934466832851766328241515210179422644395718121717021756492 ~
1964449396532222187658488244511909401340504432139858628621083179 ~
3939608443898019147873897723310286310131486955212620518278063494 ~
5711866277825659883100535155231665984394090221806314454521212978 ~
9734471488741258268223860236027109981191520568823472398358013366 ~
0683786328867928619732367253606685216856320119489780733958419190 ~
6659583867852941241871821727987506103946064819585745620060892122 ~
8416394373846549589932028481236433466119707324309545859073361878 ~
6290631850165106267576851216357588696307451999220010776676830946 ~
9814975622682434793671310841210219520899481912444048751171059184 ~
4139907889455775184621619041530934543802808938628073237578615267 ~
7971143323241969857805637630180884386640607175368321362629671224 ~
2609428540110963218262765120117022552929289655594608204938409069 ~
0760692003954646191640021567336017909631872891998634341086903200 ~
5796637103128612356988817640364252540837098108148351903121318624 ~
7228181050845123690190646632235938872454630737272808789830041018 ~
9485913673742589418124056729191238003306344998219631580386381054 ~
2457893450084553280313511884341007373060595654437362488771292628 ~
9807423539074061786905784443105274262641767830058221486462289361 ~
9296692992033046693328438158053564864073184440599549689353773183 ~
6726613130108623588021288043289344562140479789454233736058506327 ~
0439981932635916687341943656783901281912202816229500333012236091 ~
8587559201959081224153679499095448881099758919890811581163538891 ~
6339402923722049848375224236209100834097566791710084167957022331 ~
7897107102928884897013099533995424415335060625843921452433864640 ~
3432440657317477553405404481006177612569084746461432976543900008 ~
3826521145210162366431119798731902751191441213616962045693602633 ~
6102355962140467029012156796418735746835873172331004745963339773 ~
2477044918885134415363760091537564267438450166221393719306748706 ~
2881595464819775192207710236743289062690709117919412776212245117 ~
2354677115640433357720616661564674474627305622913332030953340551 ~
3841718194605321501426328000879551813296754972846701883657425342 ~

```
    5016994231069156343106626043412205213831587971115075454063290657 ~
    0248488648697402872037259869281149360627403842332874942332178578 ~
    7750735571857043787379693402336902911446961448649769719434527467 ~
    4429603089437192540526658890710662062575509930379976658367936112 ~
    8137451104971506153783743579555867972129358764463093757203221320 ~
    2460565661129971310275869112846043251843432691552928458573495971 ~
    5042565399302112184947232132380516549802909919676815118022483192 ~
    5127372199792134331067642187484426215985121676396779352982985195 ~
    8545392106957880586853123277545433229161989053189053725391582222 ~
    9232597278133427818256064882333760719681014481453198336237910767 ~
    1255017528826351836492103572587410356573894694875444694018175923 ~
    0609370828146501857425324969212764624247832210765473750568198834 ~
    5641035458027261252285503154325039591848918982630498759115406321 ~
    0354263890012837426155187877318375862355175378506956599570028011 ~
    5841258870150030170259167463020842412449128392380525772514737141 ~
    2310230172563968305553583262840383638157686828464330456805994018 ~
    7001071952092970177990583216417579868116586547147748964716547948 ~
    8312140431836079844314055731179349677763739898930227765607058530 ~
    4083747752640947435070395214524701683884070908706147194437225650 ~
    2823145872995869738316897126851939042297110721350756978037262545 ~
    8141095038270388987364516284820180468288205829135339013835649144 ~
    3004015706509887926715417450706686888783438055583501196745862340 ~
    8059532724727843829259395771584036885940989939255241688378793572 ~
    7967951654076673927031256418760962190243046993485989199060012977 ~
    7469214532970421677817261517850653008552559997940209969455431545 ~
    2745856704403686680428648404512881182309793496962721836492935516 ~
    2029872469583299481932978335803459023227052612542114437084359584 ~
    9443383638388317751841160881711251279233374577219339820819005406 ~
    3292937775306906607415304997682647124407768817248673421685881509 ~
    9133422075930947173855159340808957124410634720893194912880783576 ~
    3115829400549708918023366596077070927599010527028150868897828549 ~
    4340372642729262103487013992868853550062061514343078665396085995 ~
    0058714939141652065302070085265624074703660736605333805263766757 ~
    2018839497277047222153633851135483463624619855425993871933367482 ~
    0422097449956672702505446423243957506869591330193746919142980999 ~
    3424230550172665212092414559625960554427590951996824313084279693 ~
    7113207021049823238195747175985519501864630940297594363194450091 ~
    9150616049228764323192129703446093584259267276386814363309856853 ~
    2786024332141052330760658841495858718197071242995959226781172796 ~
    4438853796763139274314227953114500064922126500133268623021550837
    \cs_end:
  }
```

(*End definition for* `\__fp_trig_inverse_two_pi:`.)

__fp_trig_large:ww
__fp_trig_large_auxi:wwwwww
__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.

```
\cs_new:Npn \__fp_trig_large:ww #1, #2#3#4#5#6;
  {
    \exp_after:wN \__fp_trig_large_auxi:wwwwww
    \int_use:N \__int_eval:w (#1 - 32) / 64 \exp_after:wN ,
    \int_use:N \__int_eval:w (#1 - 4) / 8 \exp_after:wN ,
    \__int_value:w #1 \__fp_trig_inverse_two_pi: ;
    {#2}{#3}{#4}{#5} ;
  }
\cs_new:Npn \__fp_trig_large_auxi:wwwwww #1, #2, #3, #4!
  {
    \prg_replicate:nn {#1} { \__fp_trig_large_auxii:ww }
    \prg_replicate:nn { #2 - #1 * \c_eight }
      { \__fp_trig_large_auxiii:wNNNNNNNN }
    \prg_replicate:nn { #3 - #2 * \c_eight }
      { \__fp_trig_large_auxiv:wN }
    \prg_replicate:nn { \c_eight } { \__fp_pack_twice_four:wNNNNNNNN }
    \__fp_trig_large_auxv:www
    ;
  }
\cs_new:Npn \__fp_trig_large_auxii:ww #1; #2 ~ { #1; }
\cs_new:Npn \__fp_trig_large_auxiii:wNNNNNNNN
  #1; #2#3#4#5#6#7#8#9 { #1; }
\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.

```
\cs_new:Npn \__fp_trig_large_auxv:www #1; #2; #3;
  {
    \exp_after:wN \__fp_use_i_until_s:nw
    \exp_after:wN \__fp_trig_large_auxvii:w
    \int_use:N \__int_eval:w \c__fp_leading_shift_int
      \prg_replicate:nn { \c_thirteen }
        { \__fp_trig_large_auxvi:wnnnnnnnn }
      + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
      \__fp_use_i_until_s:nw
      ; #3 #1 ; ;
```

```
  }
\cs_new:Npn \__fp_trig_large_auxvi:wnnnnnnnn #1; #2#3#4#5#6#7#8#9
  {
    \exp_after:wN \__fp_trig_large_pack:NNNNNw
    \int_use:N \__int_eval:w \c__fp_middle_shift_int
      + #2*#9 + #3*#8 + #4*#7 + #5*#6
      #1; {#2}{#3}{#4}{#5} {#7}{#8}{#9}
  }
\cs_new:Npn \__fp_trig_large_pack:NNNNNw #1#2#3#4#5#6;
  { + #1#2#3#4#5 ; #6 }
```

(*End definition for* `\__fp_trig_large_auxv:www`*,* `\__fp_trig_large_auxvi:wnnnnnnnn`*, and* `\__fp_trig_large_pack:NNNNNw`*.*)

`\__fp_trig_large_auxvii:w`
`\__fp_trig_large_auxviii:w`
`\__fp_trig_large_auxix:Nw`
`\__fp_trig_large_auxx:wNNNNN`
`\__fp_trig_large_auxxi:w`

The `auxvii` auxiliary is followed by 52 digits and a semicolon. We find the octant as the integer part of 8 times what follows, or equivalently as the integer part of `#1#2#3`/125, and add it to the surrounding integer expression for the octant. We then compute 8 times the 52-digit number, with a minus sign if the octant is odd. Again, the last `middle` shift is converted to a `trailing` shift. Any integer part (including negative values which come up when the octant is odd) is discarded by `\__fp_use_i_until_s:nw`. The resulting fractional part should then be converted to radians by multiplying by $2\pi/8$, but first, build an extended precision number by abusing `\__fp_ep_to_ep_loop:N` with the appropriate trailing markers. Finally, `\__fp_trig_small:ww` sets up the argument for the functions which compute the Taylor series.

```
\cs_new:Npn \__fp_trig_large_auxvii:w #1#2#3
  {
    \exp_after:wN \__fp_trig_large_auxviii:ww
    \int_use:N \__int_eval:w (#1#2#3 - 62) / 125 ;
    #1#2#3
  }
\cs_new:Npn \__fp_trig_large_auxviii:ww #1;
  {
    + #1
    \if_int_odd:w #1 \exp_stop_f:
      \exp_after:wN \__fp_trig_large_auxix:Nw
      \exp_after:wN -
    \else:
      \exp_after:wN \__fp_trig_large_auxix:Nw
      \exp_after:wN +
    \fi:
  }
\cs_new_nopar:Npn \__fp_trig_large_auxix:Nw
  {
    \exp_after:wN \__fp_use_i_until_s:nw
    \exp_after:wN \__fp_trig_large_auxxi:w
    \int_use:N \__int_eval:w \c__fp_leading_shift_int
      \prg_replicate:nn { \c_thirteen }
        { \__fp_trig_large_auxx:wNNNNN }
      + \c__fp_trailing_shift_int - \c__fp_middle_shift_int
      ;
  }
```

```
\cs_new:Npn \__fp_trig_large_auxx:wNNNNN #1; #2 #3#4#5#6
  {
    \exp_after:wN \__fp_trig_large_pack:NNNNNw
    \int_use:N \__int_eval:w \c__fp_middle_shift_int
      #2 \c_eight * #3#4#5#6
      #1; #2
  }
\cs_new:Npn \__fp_trig_large_auxxi:w #1;
  {
    \exp_after:wN \__fp_ep_mul_raw:wwwwN
    \int_use:N \__int_eval:w \c_zero \__fp_ep_to_ep_loop:N #1 ; ; !
    0,{7853}{9816}{3397}{4483}{0961}{5661};
    \__fp_trig_small:ww
  }
```

(*End definition for* `\__fp_trig_large_auxvii:w` *and* `\__fp_trig_large_auxviii:w`.)

31.1.6 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.

```
\cs_new:Npn \__fp_sin_series_o:NNwwww #1#2#3. #4;
  {
    \__fp_fixed_mul:wwn #4; #4;
    {
      \exp_after:wN \__fp_sin_series_aux_o:NNnwww
```

```
      \exp_after:wN #1
      \__int_value:w
        \if_int_odd:w \__int_eval:w ( #3 + \c_two ) / \c_four \__int_eval_end:
          #2
        \else:
          \if_meaning:w #2 0 2 \else: 0 \fi:
        \fi:
      {#3}
    }
  }
\cs_new:Npn \__fp_sin_series_aux_o:NNnwww #1#2#3 #4; #5,#6;
  {
    \if_int_odd:w \__int_eval:w #3 / \c_two \__int_eval_end:
      \exp_after:wN \use_i:nn
    \else:
      \exp_after:wN \use_ii:nn
    \fi:
    { % 1/18!
      \__fp_fixed_mul_sub_back:wwwn     {0000}{0000}{0000}{0001}{5619}{2070};
                                  #4; {0000}{0000}{0000}{0477}{9477}{3324};
      \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0000}{0011}{4707}{4559}{7730};
      \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0000}{2087}{6756}{9878}{6810};
      \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0027}{5573}{1922}{3985}{8907};
      \__fp_fixed_mul_sub_back:wwwn #4; {0000}{2480}{1587}{3015}{8730}{1587};
      \__fp_fixed_mul_sub_back:wwwn #4; {0013}{8888}{8888}{8888}{8888}{8889};
      \__fp_fixed_mul_sub_back:wwwn #4; {0416}{6666}{6666}{6666}{6666}{6667};
      \__fp_fixed_mul_sub_back:wwwn #4; {5000}{0000}{0000}{0000}{0000}{0000};
      \__fp_fixed_mul_sub_back:wwwn #4;{10000}{0000}{0000}{0000}{0000}{0000};
      { \__fp_fixed_continue:wn 0, }
    }
    { % 1/17!
      \__fp_fixed_mul_sub_back:wwwn     {0000}{0000}{0000}{0028}{1145}{7254};
                                  #4; {0000}{0000}{0000}{7647}{1637}{3182};
      \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0000}{0160}{5904}{3836}{8216};
      \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0002}{5052}{1083}{8544}{1719};
      \__fp_fixed_mul_sub_back:wwwn #4; {0000}{0275}{5731}{9223}{9858}{9065};
      \__fp_fixed_mul_sub_back:wwwn #4; {0001}{9841}{2698}{4126}{9841}{2698};
      \__fp_fixed_mul_sub_back:wwwn #4; {0083}{3333}{3333}{3333}{3333}{3333};
      \__fp_fixed_mul_sub_back:wwwn #4; {1666}{6666}{6666}{6666}{6666}{6667};
      \__fp_fixed_mul_sub_back:wwwn #4;{10000}{0000}{0000}{0000}{0000}{0000};
      { \__fp_ep_mul:wwwwn 0, } #5,#6;
    }
    {
      \exp_after:wN \__fp_sanitize:Nw
      \exp_after:wN #2
      \int_use:N \__int_eval:w #1
    }
    #2
  }
```

(*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` expansion produces 0 for a positive final result, and 2 otherwise. A similar story holds for $\cot(x)$.

The auxiliary receives the sign, the octant, the square of the (reduced) input, and the (reduced) input (an extended-precision number) as arguments. It then computes the numerator and denominator of

$$\tan(x) \simeq \frac{x(1 - x^2(a_1 - x^2(a_2 - x^2(a_3 - x^2(a_4 - x^2 a_5)))))}{1 - x^2(b_1 - x^2(b_2 - x^2(b_3 - x^2(b_4 - x^2 b_5))))}.$$

The ratio is computed by `\__fp_ep_div:wwwwn`, then converted to a floating point number. For octants `#3` (really, quadrants) next to a pole of the functions, the fixed point numerator and denominator are exchanged before computing the ratio. Note that this `\if_int_odd:w` test relies on the fact that the octant is at least 1.

```
\cs_new:Npn \__fp_tan_series_o:NNwwww #1#2#3. #4;
  {
    \__fp_fixed_mul:wwn #4; #4;
    {
      \exp_after:wN \__fp_tan_series_aux_o:Nnwww
      \__int_value:w
        \if_int_odd:w \__int_eval:w #3 / \c_two \__int_eval_end:
          \exp_after:wN \reverse_if:N
        \fi:
        \if_meaning:w #1#2 2 \else: 0 \fi:
      {#3}
    }
  }
\cs_new:Npn \__fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5;
  {
    \__fp_fixed_mul_sub_back:wwwn     {0000}{0000}{1527}{3493}{0856}{7059};
                                  #3; {0000}{0159}{6080}{0274}{5257}{6472};
    \__fp_fixed_mul_sub_back:wwwn #3; {0002}{4571}{2320}{0157}{2558}{8481};
    \__fp_fixed_mul_sub_back:wwwn #3; {0115}{5830}{7533}{5397}{3168}{2147};
    \__fp_fixed_mul_sub_back:wwwn #3; {1929}{8245}{6140}{3508}{7719}{2982};
    \__fp_fixed_mul_sub_back:wwwn #3;{10000}{0000}{0000}{0000}{0000}{0000};
    { \__fp_ep_mul:wwwwn 0, } #4,#5;
    {
      \__fp_fixed_mul_sub_back:wwwn     {0000}{0007}{0258}{0681}{9408}{4706};
                                    #3; {0000}{2343}{7175}{1399}{6151}{7670};
      \__fp_fixed_mul_sub_back:wwwn #3; {0019}{2638}{4588}{9232}{8861}{3691};
      \__fp_fixed_mul_sub_back:wwwn #3; {0536}{6357}{0691}{4344}{6852}{4252};
      \__fp_fixed_mul_sub_back:wwwn #3; {5263}{1578}{9473}{6842}{1052}{6315};
      \__fp_fixed_mul_sub_back:wwwn #3;{10000}{0000}{0000}{0000}{0000}{0000};
      {
```

```
        \reverse_if:N \if_int_odd:w
            \__int_eval:w (#2 - \c_one) / \c_two \__int_eval_end:
          \exp_after:wN \__fp_reverse_args:Nww
        \fi:
        \__fp_ep_div:wwwwn 0,
      }
    }
    {
      \exp_after:wN \__fp_sanitize:Nw
      \exp_after:wN #1
      \int_use:N \__int_eval:w \__fp_ep_to_float:wwN
    }
    #1
  }
```

(*End definition for* `\__fp_tan_series_o:NNwwww` *and* `\__fp_tan_series_aux_o:Nnwww`.)

31.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arccosecant, and arcsecant) are based on a function often denoted `atan2`. This function is accessed directly by feeding two arguments to arctangent, and is defined by $\operatorname{atan}(y, x) = \operatorname{atan}(y/x)$ for generic y and x. Its advantages over the conventional arctangent is that it takes values in $[-\pi, \pi]$ rather than $[-\pi/2, \pi/2]$, and that it is better behaved in boundary cases. Other inverse trigonometric functions are expressed in terms of atan as

$$\operatorname{acos} x = \operatorname{atan}(\sqrt{1-x^2}, x) \tag{4}$$

$$\operatorname{asin} x = \operatorname{atan}(x, \sqrt{1-x^2}) \tag{5}$$

$$\operatorname{asec} x = \operatorname{atan}(\sqrt{x^2-1}, 1) \tag{6}$$

$$\operatorname{acsc} x = \operatorname{atan}(1, \sqrt{x^2-1}) \tag{7}$$

$$\operatorname{atan} x = \operatorname{atan}(x, 1) \tag{8}$$

$$\operatorname{acot} x = \operatorname{atan}(1, x). \tag{9}$$

Rather than introducing a new function, `atan2`, the arctangent function `atan` is overloaded: it can take one or two arguments. In the comments below, following many texts, we call the first argument y and the second x, because $\operatorname{atan}(y, x) = \operatorname{atan}(y/x)$ is the angular coordinate of the point (x, y).

As for direct trigonometric functions, the first step in computing $\operatorname{atan}(y, x)$ is argument reduction. The sign of y 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 $\operatorname{atan} \frac{|y|}{x}$ is given by a nicely convergent Taylor series;

1 $0 < 0.41421x < |y| < x$, then $\operatorname{atan}\frac{|y|}{x} = \frac{\pi}{4} - \operatorname{atan}\frac{x-|y|}{x+|y|}$;

2 $0 < 0.41421|y| < x < |y|$, then $\operatorname{atan}\frac{|y|}{x} = \frac{\pi}{4} + \operatorname{atan}\frac{-x+|y|}{x+|y|}$;

3 $0 < x < 0.41421|y|$, then $\operatorname{atan}\frac{|y|}{x} = \frac{\pi}{2} - \operatorname{atan}\frac{x}{|y|}$;

4 $0 < -x < 0.41421|y|$, then $\operatorname{atan}\frac{|y|}{x} = \frac{\pi}{2} + \operatorname{atan}\frac{-x}{|y|}$;

5 $0 < 0.41421|y| < -x < |y|$, then $\operatorname{atan}\frac{|y|}{x} = \frac{3\pi}{4} - \operatorname{atan}\frac{x+|y|}{-x+|y|}$;

6 $0 < -0.41421x < |y| < -x$, then $\operatorname{atan}\frac{|y|}{x} = \frac{3\pi}{4} + \operatorname{atan}\frac{-x-|y|}{-x+|y|}$;

7 $0 < |y| < -0.41421x$, then $\operatorname{atan}\frac{|y|}{x} = \pi - \operatorname{atan}\frac{|y|}{-x}$.

In the following, we 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.

31.2.1 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 $\operatorname{acot}(x) = \operatorname{atan}(1, x) = \operatorname{acot}(x, 1)$.

```
\cs_new_nopar:Npn \__fp_atan_o:Nw
  {
    \__fp_atan_dispatch_o:NNnNw
      \__fp_acotii_o:Nww \__fp_atanii_o:Nww { atan }
  }
\cs_new_nopar:Npn \__fp_acot_o:Nw
  {
    \__fp_atan_dispatch_o:NNnNw
      \__fp_atanii_o:Nww \__fp_acotii_o:Nww { acot }
  }
\cs_new:Npn \__fp_atan_dispatch_o:NNnNw #1#2#3#4#5@
  {
    \if_case:w
      \__int_eval:w \__fp_array_count:n {#5} - \c_one \__int_eval_end:
         \exp_after:wN #1 \exp_after:wN #4 \c_one_fp #5
         \tex_romannumeral:D
    \or: #2 #4 #5 \tex_romannumeral:D
    \else:
      \__msg_kernel_expandable_error:nnnnn
        { kernel } { fp-num-args } { #3() } { 1 } { 2 }
      \exp_after:wN \c_nan_fp \tex_romannumeral:D
    \fi:
    \exp_after:wN \c_zero
  }
```

(*End definition for* `\__fp_atan_o:Nw` *and* `\__fp_acot_o:Nw`.)

`\__fp_atanii_o:Nww`
`\__fp_acotii_o:Nww`

If either operand is `nan`, we return it. If both are normal, we call `\__fp_atan_normal_o:NNnwNnw`. If both are zero or both infinity, we call `\__fp_atan_inf_o:NNNw` with argument 2, leading to a result among $\{\pm\pi/4, \pm 3\pi/4\}$ (in degrees, $\{\pm 45, \pm 135\}$). Otherwise, one is much bigger than the other, and we call `\__fp_atan_inf_o:NNNw` with either an argument of 4, leading to the values $\pm\pi/2$ (in degrees, ± 90), or 0, leading to $\{\pm 0, \pm\pi\}$ (in degrees, $\{\pm 0, \pm 180\}$). Since $\operatorname{acot}(x, y) = \operatorname{atan}(y, x)$, `\__fp_acotii_o:ww` simply reverses its two arguments.

```
\cs_new:Npn \__fp_atanii_o:Nww
    #1 \s__fp \__fp_chk:w #2#3#4; \s__fp \__fp_chk:w #5
  {
    \if_meaning:w 3 #2 \__fp_case_return_i_o:ww \fi:
    \if_meaning:w 3 #5 \__fp_case_return_ii_o:ww \fi:
    \if_case:w
      \if_meaning:w #2 #5
        \if_meaning:w 1 #2 \c_ten \else: \c_zero \fi:
      \else:
        \if_int_compare:w #2 > #5 \c_one \else: \c_two \fi:
      \fi:
         \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 \c_two }
    \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 \c_four }
    \or: \__fp_case_return:nw { \__fp_atan_inf_o:NNNw #1 #3 \c_zero }
    \fi:
    \__fp_atan_normal_o:NNnwNnw #1
    \s__fp \__fp_chk:w #2#3#4;
    \s__fp \__fp_chk:w #5
  }
\cs_new:Npn \__fp_acotii_o:Nww #1#2; #3;
  { \__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`; $\operatorname{atan} z/z = 1$ as a fixed point number; $z = 0$ as a fixed point number; and $z = 0$ as an extended-precision number. Given the values we provide, $\operatorname{atan} z$ will be computed to be 0, and the result will be $[\texttt{#3}/2] \cdot \pi/4$ if the sign `#5` of x is positive, and $[(7 - \texttt{#3})/2] \cdot \pi/4$ for negative x, where the divisions are rounded up.

```
\cs_new:Npn \__fp_atan_inf_o:NNNw #1#2#3 \s__fp \__fp_chk:w #4#5#6;
  {
    \exp_after:wN \__fp_atan_combine_o:NwwwwwN
    \exp_after:wN #2
    \int_use:N \__int_eval:w
      \if_meaning:w 2 #5 \c_seven - \fi: #3 \exp_after:wN ;
    \c__fp_one_fixed_tl ;
    {0000}{0000}{0000}{0000}{0000}{0000};
    0,{0000}{0000}{0000}{0000}{0000}{0000}; #1
  }
```

(*End definition for* `\__fp_atan_inf_o:NNNw`.)

`\__fp_atan_normal_o:NNnwNnw`

Here we simply reorder the floating point data into a pair of signed extended-precision numbers, that is, a sign, an exponent ending with a comma, and a six-block mantissa ending with a semi-colon. This extended precision is required by other inverse trigonometric functions, to compute things like $\operatorname{atan}(x, \sqrt{1-x^2})$ without intermediate rounding errors.

```
\cs_new_protected:Npn \__fp_atan_normal_o:NNnwNnw
    #1 \s__fp \__fp_chk:w 1#2#3#4; \s__fp \__fp_chk:w 1#5#6#7;
  {
    \__fp_atan_test_o:NwwNwwN
      #2 #3, #4{0000}{0000};
      #5 #6, #7{0000}{0000}; #1
  }
```

(*End 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.

```
\cs_new:Npn \__fp_atan_test_o:NwwNwwN #1#2,#3; #4#5,#6;
  {
    \exp_after:wN \__fp_atan_combine_o:NwwwwwN
    \exp_after:wN #1
    \int_use:N \__int_eval:w
      \if_meaning:w 2 #4
        \c_seven - \__int_eval:w
      \fi:
      \if_int_compare:w
          \__fp_ep_compare:wwww #2,#3; #5,#6; > \c_zero
        \c_three -
        \exp_after:wN \__fp_reverse_args:Nww
      \fi:
      \__fp_atan_div:wnwwnw #2,#3; #5,#6;
  }
```

(*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 atan $\frac{a}{b}$. In any case, call __fp_atan_auxi:ww followed by z, as a comma-delimited exponent and a fixed point number.

```
\cs_new:Npn \__fp_atan_div:wnwwnw #1,#2#3; #4,#5#6;
  {
    \if_int_compare:w
      \__int_eval:w 41421 * #5 < #2 000
        \if_case:w \__int_eval:w #4 - #1 \__int_eval_end: 00 \or: 0 \fi:
      \exp_stop_f:
      \exp_after:wN \__fp_atan_near:wwwn
    \fi:
    \c_zero
    \__fp_ep_div:wwwwn #1,{#2}#3; #4,{#5}#6;
    \__fp_atan_auxi:ww
  }
\cs_new:Npn \__fp_atan_near:wwwn
    \c_zero \__fp_ep_div:wwwwn #1,#2; #3,
  {
    \c_one
    \__fp_ep_to_fixed:wwn #1 - #3, #2;
    \__fp_atan_near_aux:wwn
  }
\cs_new:Npn \__fp_atan_near_aux:wwn #1; #2;
  {
    \__fp_fixed_add:wwn #1; #2;
    { \__fp_fixed_sub:wwn #2; #1; { \__fp_ep_div:wwwwn 0, } 0, }
  }
```

(*End definition for* __fp_atan_div:wnwwnw *and* __fp_atan_near:wwwn.)

__fp_atan_auxi:ww
__fp_atan_auxii:w

Convert z from a representation as an exponent and a fixed point number in $[0.01, 1)$ to a fixed point number only, then set up the call to __fp_atan_Taylor_loop:www, followed by the fixed point representation of z and the old representation.

```
\cs_new:Npn \__fp_atan_auxi:ww #1,#2;
  { \__fp_ep_to_fixed:wwn #1,#2; \__fp_atan_auxii:w #1,#2; }
\cs_new:Npn \__fp_atan_auxii:w #1;
  {
    \__fp_fixed_mul:wwn #1; #1;
    {
      \__fp_atan_Taylor_loop:www 39 ;
        {0000}{0000}{0000}{0000}{0000}{0000} ;
    }
    ! #1;
  }
```

(*End definition for* __fp_atan_auxi:ww *and* __fp_atan_auxii:w.)

__fp_atan_Taylor_loop:www
__fp_atan_Taylor_break:w

We compute the series of $(\operatorname{atan} z)/z$. A typical intermediate stage has #1 $= 2k - 1$, #2 $= \frac{1}{2k+1} - z^2(\frac{1}{2k+3} - z^2(\cdots - z^2\frac{1}{39}))$, and #3 $= z^2$. To go to the next step $k \to k - 1$, we compute $\frac{1}{2k-1}$, then subtract from it z^2 times #2. The loop stops when $k = 0$: then

#2 is $(\operatorname{atan} z)/z$, and there is a need to clean up all the unnecessary data, end the integer expression computing the octant with a semicolon, and leave the result #2 afterwards.

```
\cs_new:Npn \__fp_atan_Taylor_loop:www #1; #2; #3;
  {
    \if_int_compare:w #1 = \c_minus_one
      \__fp_atan_Taylor_break:w
    \fi:
    \exp_after:wN \__fp_fixed_div_int:wwN \c__fp_one_fixed_tl ; #1;
    \__fp_rrot:www \__fp_fixed_mul_sub_back:wwwn #2; #3;
    {
      \exp_after:wN \__fp_atan_Taylor_loop:www
      \int_use:N \__int_eval:w #1 - \c_two ;
    }
    #3;
  }
\cs_new:Npn \__fp_atan_Taylor_break:w
    \fi: #1 \__fp_fixed_mul_sub_back:wwwn #2; #3 !
  { \fi: ; #2 ; }
```

(*End definition for* `\__fp_atan_Taylor_loop:www` *and* `\__fp_atan_Taylor_break:w`.)

`\__fp_atan_combine_o:NwwwwwN`
`\__fp_atan_combine_aux:ww`

This receives a ⟨*sign*⟩, an ⟨*octant*⟩, a fixed point value of $(\operatorname{atan} z)/z$, a fixed point number z, and another representation of z, as an ⟨*exponent*⟩ and the fixed point number $10^{-\langle exponent\rangle} z$, 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

$$\langle sign\rangle \left(\left\lceil \frac{\langle octant\rangle}{2} \right\rceil \frac{\pi}{4} + (-1)^{\langle octant\rangle} \frac{\operatorname{atan} z}{z} \cdot z \right) , \tag{10}$$

multiplied by $180/\pi$ if working in degrees, and using in any case the most appropriate representation of z. The floating point result is passed to `\__fp_sanitize:Nw`, which checks for overflow or underflow. If the octant is 0, leave the exponent #5 for `\__fp_sanitize:Nw`, and multiply #3 $= \frac{\operatorname{atan} z}{z}$ with #6, the adjusted z. Otherwise, multiply #3 $= \frac{\operatorname{atan} z}{z}$ with #4 $= z$, then compute the appropriate multiple of $\frac{\pi}{4}$ and add or subtract the product #3 $\cdot$ #4. In both cases, convert to a floating point with `\__fp_fixed_to_float:wN`.

```
\cs_new:Npn \__fp_atan_combine_o:NwwwwwN #1 #2; #3; #4; #5,#6; #7
  {
    \exp_after:wN \__fp_sanitize:Nw
    \exp_after:wN #1
    \int_use:N \__int_eval:w
      \if_meaning:w 0 #2
        \exp_after:wN \use_i:nn
      \else:
        \exp_after:wN \use_ii:nn
      \fi:
      { #5 \__fp_fixed_mul:wwn #3; #6; }
      {
        \__fp_fixed_mul:wwn #3; #4;
```

```
        {
          \exp_after:wN \__fp_atan_combine_aux:ww
          \int_use:N \__int_eval:w #2 / \c_two ; #2;
        }
      }
      { #7 \__fp_fixed_to_float:wN \__fp_fixed_to_float_rad:wN }
      #1
  }
\cs_new:Npn \__fp_atan_combine_aux:ww #1; #2;
  {
    \__fp_fixed_mul_short:wwn
      {7853}{9816}{3397}{4483}{0961}{5661};
      {#1}{0000}{0000};
    {
      \if_int_odd:w #2 \exp_stop_f:
        \exp_after:wN \__fp_fixed_sub:wwn
      \else:
        \exp_after:wN \__fp_fixed_add:wwn
      \fi:
    }
  }
```

(*End definition for* `\__fp_atan_combine_o:NwwwwwN` *and* `\__fp_atan_combine_aux:ww`.)

31.2.2 Arcsine and arccosine

`\__fp_asin_o:w` Again, the first argument provided by l3fp-parse is `\use_i:nn` if we are to work in radians and `\use_ii:nn` for degrees. Then comes a floating point number. The arcsine of ± 0 or `nan` is the same floating point number. The arcsine of $\pm\infty$ raises an invalid operation exception. Otherwise, call an auxiliary common with `\__fp_acos_o:w`, feeding it information about what function is being performed (for "invalid operation" exceptions).

```
\cs_new:Npn \__fp_asin_o:w #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_case:w #2 \exp_stop_f:
      \__fp_case_return_same_o:w
    \or:
      \__fp_case_use:nw
        { \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { asin } { asind } } }
    \or:
      \__fp_case_use:nw
        { \__fp_invalid_operation_o:fw { #1 { asin } { asind } } }
    \else:
      \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3;
  }
```

(*End definition for* `\__fp_asin_o:w`.)

`\__fp_acos_o:w` The arccosine of ± 0 is $\pi/2$ (in degrees, 90). The arccosine of $\pm\infty$ raises an invalid operation exception. The arccosine of `nan` is itself. Otherwise, call an auxiliary common with __fp_sin_o:w, informing it that it was called by acos or acosd, and preparing to swap some arguments down the line.

```
\cs_new:Npn \__fp_acos_o:w #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_case:w #2 \exp_stop_f:
      \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 \c_four }
    \or:
      \__fp_case_use:nw
        {
          \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { acos } { acosd } }
            \__fp_reverse_args:Nww
        }
    \or:
      \__fp_case_use:nw
        { \__fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
    \else:
      \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3;
  }
```

(*End 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.

```
\cs_new:Npn \__fp_asin_normal_o:NfwNnnnnw
    #1#2#3 \s__fp \__fp_chk:w 1#4#5#6#7#8#9;
  {
    \if_int_compare:w #5 < \c_one
      \exp_after:wN \__fp_use_none_until_s:w
    \fi:
    \if_int_compare:w \__int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
      \exp_after:wN \__fp_use_none_until_s:w
    \fi:
    \__fp_use_i:ww
    \__fp_invalid_operation_o:fw {#2}
      \s__fp \__fp_chk:w 1#4{#5}{#6}{#7}{#8}{#9};
    \__fp_asin_auxi_o:NnNww
      #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000};
  }
```

(*End definition for* __fp_asin_normal_o:NfwNnnnnw.)

__fp_asin_auxi_o:NnNww
__fp_asin_isqrt:wn

We compute $x/\sqrt{1-x^2}$. This function is used by asin and acos, but also by acsc and asec after inverting the operand, thus it must manipulate extended-precision numbers. First evaluate $1-x^2$ as $(1+x)(1-x)$: this behaves better near $x=1$. We do the addition/subtraction with fixed point numbers (they are not implemented for extended-precision floats), but go back to extended-precision floats to multiply and compute the inverse square root $1/\sqrt{1-x^2}$. Finally, multiply by the (positive) extended-precision float $|x|$, and feed the (signed) result, and the number $+1$, as arguments to the arctangent function. When computing the arccosine, the arguments $x/\sqrt{1-x^2}$ and $+1$ are swapped by `#2` (`\__fp_reverse_args:Nww` in that case) before `\__fp_atan_test_o:NwwNwwN` is evaluated. Note that the arctangent function requires normalized arguments, hence the need for `ep_to_ep` and `continue` after `ep_mul`.

```
\cs_new:Npn \__fp_asin_auxi_o:NnNww #1#2#3#4,#5;
  {
    \__fp_ep_to_fixed:wwn #4,#5;
    \__fp_asin_isqrt:wn
    \__fp_ep_mul:wwwwn #4,#5;
    \__fp_ep_to_ep:wwN
    \__fp_fixed_continue:wn
    { #2 \__fp_atan_test_o:NwwNwwN #3 }
    0 1,{1000}{0000}{0000}{0000}{0000}{0000}; #1
  }
\cs_new:Npn \__fp_asin_isqrt:wn #1;
  {
    \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl ; #1;
    {
      \__fp_fixed_add_one:wN #1;
      \__fp_fixed_continue:wn { \__fp_ep_mul:wwwwn 0, } 0,
    }
    \__fp_ep_isqrt:wwn
  }
```

(*End definition for* `\__fp_asin_auxi_o:NnNww` *and* `\__fp_asin_isqrt:wn`.)

31.2.3 Arccosecant and arcsecant

`\__fp_acsc_o:w` Cases are mostly labelled by `#2`, except when `#2` is 2: then we use `#3#2`, which is $02 = 2$ when the number is $+\infty$ and 22 when the number is $-\infty$. The arccosecant of ± 0 raises an invalid operation exception. The arccosecant of $\pm\infty$ is ± 0 with the same sign. The arcosecant of `nan` is itself. Otherwise, `\__fp_acsc_normal_o:NfwNnw` does some more tests, keeping the function name (`acsc` or `acscd`) as an argument for invalid operation exceptions.

```
\cs_new:Npn \__fp_acsc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
  {
    \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
           \__fp_case_use:nw
             { \__fp_invalid_operation_o:fw { #1 { acsc } { acscd } } }
    \or:   \__fp_case_use:nw
             { \__fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acscd } } }
    \or:   \__fp_case_return_o:Nw \c_zero_fp
    \or:   \__fp_case_return_same_o:w
    \else: \__fp_case_return_o:Nw \c_minus_zero_fp
    \fi:
```

```
    \s__fp \__fp_chk:w #2 #3 #4;
  }
```

(*End definition for* `\__fp_acsc_o:w`.)

`\__fp_asec_o:w` The arcsecant of ± 0 raises an invalid operation exception. The arcsecant of $\pm\infty$ is $\pi/2$ (in degrees, 90). The arcosecant of `nan` is itself. Otherwise, do some more tests, keeping the function name `asec` (or `asecd`) as an argument for invalid operation exceptions, and a `\__fp_reverse_args:Nww` following precisely that appearing in `\__fp_acos_o:w`.

```
\cs_new:Npn \__fp_asec_o:w #1 \s__fp \__fp_chk:w #2#3; @
  {
    \if_case:w #2 \exp_stop_f:
      \__fp_case_use:nw
        { \__fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
    \or:
      \__fp_case_use:nw
        {
          \__fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
            \__fp_reverse_args:Nww
        }
    \or:   \__fp_case_use:nw { \__fp_atan_inf_o:NNNw #1 0 \c_four }
    \else: \__fp_case_return_same_o:w
    \fi:
    \s__fp \__fp_chk:w #2 #3;
  }
```

(*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:nNww` (with all the appropriate arguments). This computes what we want thanks to $\operatorname{acsc}(x) = \operatorname{asin}(1/x)$ and $\operatorname{asec}(x) = \operatorname{acos}(1/x)$.

```
\cs_new:Npn \__fp_acsc_normal_o:NfwNnw #1#2#3 \s__fp \__fp_chk:w 1#4#5#6;
  {
    \int_compare:nNnTF {#5} < \c_one
      {
        \__fp_invalid_operation_o:fw {#2}
          \s__fp \__fp_chk:w 1#4{#5}#6;
      }
      {
        \__fp_ep_div:wwwwn
          1,{1000}{0000}{0000}{0000}{0000}{0000};
          #5,#6{0000}{0000};
        { \__fp_asin_auxi_o:NnNww #1 {#3} #4 }
      }
  }
```

(*End definition for* `\__fp_acsc_normal_o:NfwNnw`.)

```
⟨/initex | package⟩
```

32 l3fp-convert implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

32.1 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.

```
\cs_new:Npn \__fp_trim_zeros:w #1 ;
  {
    \__fp_trim_zeros_loop:w #1
      ; \__fp_trim_zeros_loop:w 0; \__fp_trim_zeros_dot:w .; \s__stop
  }
\cs_new:Npn \__fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
\cs_new:Npn \__fp_trim_zeros_dot:w #1 .; { \__fp_trim_zeros_end:w #1 ; }
\cs_new:Npn \__fp_trim_zeros_end:w #1 ; #2 \s__stop { #1 }
```

(*End definition for* __fp_trim_zeros:w.)

32.2 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.

```
\cs_new:Npn \fp_to_scientific:N #1
  { \exp_after:wN \__fp_to_scientific_dispatch:w #1 }
\cs_generate_variant:Nn \fp_to_scientific:N { c }
\cs_new_nopar:Npn \fp_to_scientific:n
  {
    \exp_after:wN \__fp_to_scientific_dispatch:w
    \tex_romannumeral:D -‘0 \__fp_parse:n
  }
```

(*End definition for* \fp_to_scientific:N*,* \fp_to_scientific:c*, and* \fp_to_scientific:n*. These functions are documented on page ??.*)

__fp_to_scientific_dispatch:w
__fp_to_scientific_normal:wnnnnn
__fp_to_scientific_normal:wNw

Expressing an internal floating point number in scientific notation is quite easy: no rounding, and the format is very well defined. First cater for the sign: negative numbers (#2 = 2) start with -; we then only need to care about positive numbers and nan. Then filter the special cases: ±0 are represented as 0; infinities are converted to a number slightly larger than the largest after an "invalid_operation" exception; nan is represented as 0 after an "invalid_operation" exception. In the normal case, decrement the exponent and unbrace the 4 brace groups, then in a second step grab the first digit (previously hidden in braces) to order the various parts correctly. Finally trim zeros. The whole construction is within a call to \tl_to_lowercase:n, responsible for creating e with category "other".

```
\group_begin:
\char_set_catcode_other:N E
```

```
\tl_to_lowercase:n
  {
    \group_end:
    \cs_new:Npn \__fp_to_scientific_dispatch:w \s__fp \__fp_chk:w #1#2
      {
        \if_meaning:w 2 #2 \exp_after:wN - \tex_romannumeral:D -`0 \fi:
        \if_case:w #1 \exp_stop_f:
             \__fp_case_return:nw { 0 }
        \or: \exp_after:wN \__fp_to_scientific_normal:wnnnnn
        \or:
          \__fp_case_use:nw
            {
              \__fp_invalid_operation:nnw
                {
                  \exp_after:wN 1
                  \exp_after:wN E
                  \int_use:N \c__fp_max_exponent_int
                }
                { fp_to_scientific }
            }
        \or:
          \__fp_case_use:nw
            {
              \__fp_invalid_operation:nnw
                { 0 }
                { fp_to_scientific }
            }
        \fi:
        \s__fp \__fp_chk:w #1 #2
      }
    \cs_new:Npn \__fp_to_scientific_normal:wnnnnn
        \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
      {
        \if_int_compare:w #2 = \c_one
          \exp_after:wN \__fp_to_scientific_normal:wNw
        \else:
          \exp_after:wN \__fp_to_scientific_normal:wNw
          \exp_after:wN E
          \int_use:N \__int_eval:w #2 - \c_one
        \fi:
        ; #3 #4 #5 #6 ;
      }
  }
\cs_new:Npn \__fp_to_scientific_normal:wNw #1 ; #2#3;
  { \__fp_trim_zeros:w #2.#3 ; #1 }
```

(*End definition for* `\__fp_to_scientific_dispatch:w`*,* `\__fp_to_scientific_normal:wnnnnn`*, and* `\__fp_to_scientific_nor`

32.3 Decimal representation

\fp_to_decimal:N
\fp_to_decimal:c
\fp_to_decimal:n

All three public variants are based on the same __fp_to_decimal_dispatch:w after evaluating their argument to an internal floating point.

```
\cs_new:Npn \fp_to_decimal:N #1
  { \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
\cs_generate_variant:Nn \fp_to_decimal:N { c }
\cs_new_nopar:Npn \fp_to_decimal:n
  {
    \exp_after:wN \__fp_to_decimal_dispatch:w
    \tex_romannumeral:D -`0 \__fp_parse:n
  }
```

(*End definition for* \fp_to_decimal:N*,* \fp_to_decimal:c*, and* \fp_to_decimal:n*. These functions are documented on page* ??.)

__fp_to_decimal_dispatch:w
__fp_to_decimal_normal:wnnnnn
__fp_to_decimal_large:Nnnw
__fp_to_decimal_huge:wnnnn

The structure is similar to __fp_to_scientific_dispatch:w. Insert - for negative numbers. Zero gives 0, $\pm\infty$ and **nan** yield an "invalid operation" exception; note that $\pm\infty$ produces a very large output, which we don't expand now since it most likely won't be needed. Normal numbers with an exponent in the range $[1, 15]$ have that number of digits before the decimal separator: "decimate" them, and remove leading zeros with __int_value:w, then trim trailing zeros and dot. Normal numbers with an exponent 16 or larger have no decimal separator, we only need to add trailing zeros. When the exponent is non-positive, the result should be 0.⟨*zeros*⟩⟨*digits*⟩, trimmed.

```
\cs_new:Npn \__fp_to_decimal_dispatch:w \s__fp \__fp_chk:w #1#2
  {
    \if_meaning:w 2 #2 \exp_after:wN - \tex_romannumeral:D -`0 \fi:
    \if_case:w #1 \exp_stop_f:
         \__fp_case_return:nw { 0 }
    \or: \exp_after:wN \__fp_to_decimal_normal:wnnnnn
    \or:
      \__fp_case_use:nw
        {
          \__fp_invalid_operation:nnw
            {
              \exp_after:wN \exp_after:wN \exp_after:wN 1
              \prg_replicate:nn \c__fp_max_exponent_int 0
            }
            { fp_to_decimal }
        }
    \or:
      \__fp_case_use:nw
        {
          \__fp_invalid_operation:nnw
            { 0 }
            { fp_to_decimal }
        }
    \fi:
    \s__fp \__fp_chk:w #1 #2
  }
```

```
\cs_new:Npn \__fp_to_decimal_normal:wnnnnn
    \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
  {
    \int_compare:nNnTF {#2} > \c_zero
      {
        \int_compare:nNnTF {#2} < \c_sixteen
          {
            \__fp_decimate:nNnnnn { \c_sixteen - #2 }
              \__fp_to_decimal_large:Nnnw
          }
          {
            \exp_after:wN \exp_after:wN
            \exp_after:wN \__fp_to_decimal_huge:wnnnn
            \prg_replicate:nn { #2 - \c_sixteen } { 0 } ;
          }
        {#3} {#4} {#5} {#6}
      }
      {
        \exp_after:wN \__fp_trim_zeros:w
        \exp_after:wN 0
        \exp_after:wN .
        \tex_romannumeral:D -`0 \prg_replicate:nn { - #2 } { 0 }
        #3#4#5#6 ;
      }
  }
\cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4;
  {
    \exp_after:wN \__fp_trim_zeros:w \__int_value:w
      \if_int_compare:w #2 > \c_zero
        #2
      \fi:
      \exp_stop_f:
      #3.#4 ;
  }
\cs_new:Npn \__fp_to_decimal_huge:wnnnn #1; #2#3#4#5 { #2#3#4#5 #1 }
```

(*End definition for* `\__fp_to_decimal_dispatch:w` *and others.*)

32.4 Token list representation

`\fp_to_tl:N`
`\fp_to_tl:c`
`\fp_to_tl:n`

These three public functions evaluate their argument, then pass it to `\__fp_to_tl_dispatch:w`.

```
\cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \__fp_to_tl_dispatch:w #1 }
\cs_generate_variant:Nn \fp_to_tl:N { c }
\cs_new_nopar:Npn \fp_to_tl:n
  {
    \exp_after:wN \__fp_to_tl_dispatch:w
    \tex_romannumeral:D -`0 \__fp_parse:n
  }
```

(*End definition for* `\fp_to_tl:N`*,* `\fp_to_tl:c`*, and* `\fp_to_tl:n`*. These functions are documented on page* **??***.*)

`\__fp_to_tl_dispatch:w`
`\__fp_to_tl_normal:nnnnn`

A structure similar to `\__fp_to_scientific_dispatch:w` and `\__fp_to_decimal_-dispatch:w`, but without the "invalid operation" exception. First filter special cases. We express normal numbers in decimal notation if the exponent is in the range $[-2, 16]$, and otherwise use scientific notation.

```
\cs_new:Npn \__fp_to_tl_dispatch:w \s__fp \__fp_chk:w #1#2
  {
    \if_meaning:w 2 #2 \exp_after:wN - \tex_romannumeral:D -`0 \fi:
    \if_case:w #1 \exp_stop_f:
           \__fp_case_return:nw { 0 }
    \or:   \exp_after:wN \__fp_to_tl_normal:nnnnn
    \or:   \__fp_case_return:nw { \tl_to_str:n {inf} }
    \else: \__fp_case_return:nw { \tl_to_str:n {nan} }
    \fi:
  }
\cs_new:Npn \__fp_to_tl_normal:nnnnn #1
  {
    \if_int_compare:w #1 > \c_sixteen
      \exp_after:wN \__fp_to_scientific_normal:wnnnnn
    \else:
      \if_int_compare:w #1 < - \c_two
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_to_scientific_normal:wnnnnn
      \else:
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_to_decimal_normal:wnnnnn
      \fi:
    \fi:
    \s__fp \__fp_chk:w 1 0 {#1}
  }
```

(*End definition for* `\__fp_to_tl_dispatch:w` *and* `\__fp_to_tl_normal:nnnnn`*.*)

32.5 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.

32.6 Convert to dimension or integer

`\fp_to_dim:N`
`\fp_to_dim:c`
`\fp_to_dim:n`

These three public functions rely on `\fp_to_decimal:n` internally. We make sure to produce `pt` with category other.

```
\cs_new:Npx \fp_to_dim:N #1
  { \exp_not:N \fp_to_decimal:N #1 \tl_to_str:n {pt} }
\cs_generate_variant:Nn \fp_to_dim:N { c }
\cs_new:Npx \fp_to_dim:n #1
  { \exp_not:N \fp_to_decimal:n {#1} \tl_to_str:n {pt} }
```

(*End definition for* `\fp_to_dim:N`*,* `\fp_to_dim:c`*, and* `\fp_to_dim:n`*. These functions are documented on page* ??*.*)

\fp_to_int:N
\fp_to_int:c
\fp_to_int:n

These three public functions evaluate their argument, then pass it to `\fp_to_int_dispatch:w`.

```
\cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \__fp_to_int_dispatch:w #1 }
\cs_generate_variant:Nn \fp_to_int:N { c }
\cs_new_nopar:Npn \fp_to_int:n
  {
    \exp_after:wN \__fp_to_int_dispatch:w
    \tex_romannumeral:D -`0 \__fp_parse:n
  }
```

(*End definition for* `\fp_to_int:N`*,* `\fp_to_int:c`*, and* `\fp_to_int:n`*. These functions are documented on page* ??*.*)

__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.

```
\cs_new:Npn \__fp_to_int_dispatch:w #1;
  {
    \exp_after:wN \__fp_to_decimal_dispatch:w \tex_romannumeral:D -`0
    \__fp_round:Nwn \__fp_round_to_nearest:NNN #1; { 0 }
  }
```

(*End definition for* `\__fp_to_int_dispatch:w`*.*)

32.7 Convert from a dimension

\dim_to_fp:n
__fp_from_dim_test:ww
__fp_from_dim:wNw
__fp_from_dim:wNNnnnnnn
__fp_from_dim:wnnnnwNw

The dimension expression (which can in fact be a glue expression) is evaluated, converted to a number (*i.e.*, expressed in scaled points), then multiplied by $2^{-16} = 0.0000152587890625$ to give a value expressed in points. The auxiliary `\__fp_mul_npos_o:Nww` expects the desired ⟨*final sign*⟩ and two floating point operands (of the form `\s__fp` ... `;`) as arguments. This set of functions is also used to convert dimension registers to floating points while parsing expressions: in this context there is an additional exponent, which is the first argument of `\__fp_from_dim_test:ww`, and is combined with the exponent -4 of 2^{-16}. There is also a need to expand afterwards: this is performed by `\__fp_mul_npos_o:Nww`, and cancelled by `\prg_do_nothing:` in `\dim_to_fp:n`.

```
\cs_new:Npn \dim_to_fp:n #1
  {
    \exp_after:wN \__fp_from_dim_test:ww
    \exp_after:wN 0
    \exp_after:wN ,
    \__int_value:w \etex_glueexpr:D #1 ;
  }
\cs_new:Npn \__fp_from_dim_test:ww #1, #2
  {
    \if_meaning:w 0 #2
      \__fp_case_return:nw { \exp_after:wN \c_zero_fp }
    \else:
```

```
      \exp_after:wN \__fp_from_dim:wNw
      \int_use:N \__int_eval:w #1 - \c_four
        \if_meaning:w - #2
          \exp_after:wN , \exp_after:wN 2 \__int_value:w
        \else:
          \exp_after:wN , \exp_after:wN 0 \__int_value:w #2
        \fi:
    \fi:
  }
\cs_new:Npn \__fp_from_dim:wNw #1,#2#3;
  {
    \__fp_pack_twice_four:wNNNNNNNN \__fp_from_dim:wNNnnnnnn ;
    #3 000 0000 00 {10}987654321; #2 {#1}
  }
\cs_new:Npn \__fp_from_dim:wNNnnnnnn #1; #2#3#4#5#6#7#8#9
  { \__fp_from_dim:wnnnnwNn #1 {#2#300} {0000} ; }
\cs_new:Npn \__fp_from_dim:wnnnnwNn #1; #2#3#4#5#6; #7#8
  {
    \__fp_mul_npos_o:Nww #7
      \s__fp \__fp_chk:w 1 #7 {#5} #1 ;
      \s__fp \__fp_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} ;
      \prg_do_nothing:
  }
```

(*End definition for* `\dim_to_fp:n`. *This function is documented on page* 185.)

32.8 Use and eval

\fp_use:N
\fp_use:c
\fp_eval:n

Those public functions are simple copies of the decimal conversions.

```
\cs_new_eq:NN \fp_use:N \fp_to_decimal:N
\cs_generate_variant:Nn \fp_use:N { c }
\cs_new_eq:NN \fp_eval:n \fp_to_decimal:n
```

(*End definition for* `\fp_use:N`, `\fp_use:c`, *and* `\fp_eval:n`. *These functions are documented on page* 172.)

\fp_abs:n

Trivial but useful. See the implementation of `\fp_add:Nn` for an explanation of why to use `\__fp_parse:n`, namely, for better error reporting.

```
\cs_new:Npn \fp_abs:n #1
  { \fp_to_decimal:n { abs \__fp_parse:n {#1} } }
```

(*End definition for* `\fp_abs:n`. *This function is documented on page* 185.)

\fp_max:nn
\fp_min:nn

Similar to `\fp_abs:n`, for consistency with `\int_max:nn`, *etc.*

```
\cs_new:Npn \fp_max:nn #1#2
  { \fp_to_decimal:n { max ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
\cs_new:Npn \fp_min:nn #1#2
  { \fp_to_decimal:n { min ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
```

(*End definition for* `\fp_max:nn` *and* `\fp_min:nn`. *These functions are documented on page* 185.)

32.9 Convert an array of floating points to a comma list

__fp_array_to_clist:n
__fp_array_to_clist_loop:Nw

Converts an array of floating point numbers to a comma-list. If speed here ends up irrelevant, we can simplify the code for the auxiliary to become

```
\cs_new:Npn \__fp_array_to_clist_loop:Nw #1#2;
  {
    \use_none:n #1
    { , ~ } \fp_to_tl:n { #1 #2 ; }
    \__fp_array_to_clist_loop:Nw
  }
```

The \use_ii:nn function is expanded after __fp_expand:n is done, and it removes ,~ from the start of the representation.

```
\cs_new:Npn \__fp_array_to_clist:n #1
  {
    \tl_if_empty:nF {#1}
      {
        \__fp_expand:n
          {
            { \use_ii:nn }
            \__fp_array_to_clist_loop:Nw #1 { ? \__prg_break: } ;
            \__prg_break_point:
          }
      }
  }
\cs_new:Npx \__fp_array_to_clist_loop:Nw #1#2;
  {
    \exp_not:N \use_none:n #1
    \exp_not:N \exp_after:wN
                {
    \exp_not:N     \exp_after:wN ,
    \exp_not:N     \exp_after:wN \c_space_tl
    \exp_not:N     \tex_romannumeral:D -`0
    \exp_not:N     \__fp_to_tl_dispatch:w #1 #2 ;
                }
    \exp_not:N \__fp_array_to_clist_loop:Nw
  }
```

(*End definition for __fp_array_to_clist:n.*)

```
⟨/initex | package⟩
```

33 l3fp-assign implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

33.1 Assigning values

\fp_new:N Floating point variables are initialized to be +0.

```
\cs_new_protected:Npn \fp_new:N #1
  { \cs_new_eq:NN #1 \c_zero_fp }
\cs_generate_variant:Nn \fp_new:N {c}
```

(*End definition for* `\fp_new:N`*. This function is documented on page 171.*)

\fp_set:Nn
\fp_set:cn
\fp_gset:Nn
\fp_gset:cn
\fp_const:Nn
\fp_const:cn

Simply use `\__fp_parse:n` within various `f`-expanding assignments.

```
\cs_new_protected:Npn \fp_set:Nn   #1#2
  { \tl_set:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\cs_new_protected:Npn \fp_gset:Nn  #1#2
  { \tl_gset:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\cs_new_protected:Npn \fp_const:Nn #1#2
  { \tl_const:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\cs_generate_variant:Nn \fp_set:Nn {c}
\cs_generate_variant:Nn \fp_gset:Nn {c}
\cs_generate_variant:Nn \fp_const:Nn {c}
```

(*End definition for* `\fp_set:Nn` *and others. These functions are documented on page* **??**.)

\fp_set_eq:NN
\fp_set_eq:cN
\fp_set_eq:Nc
\fp_set_eq:cc
\fp_gset_eq:NN
\fp_gset_eq:cN
\fp_gset_eq:Nc
\fp_gset_eq:cc

Copying a floating point is the same as copying the underlying token list.

```
\cs_new_eq:NN \fp_set_eq:NN  \tl_set_eq:NN
\cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
\cs_generate_variant:Nn \fp_set_eq:NN  { c , Nc , cc }
\cs_generate_variant:Nn \fp_gset_eq:NN { c , Nc , cc }
```

(*End definition for* `\fp_set_eq:NN` *and others. These functions are documented on page* **??**.)

\fp_zero:N
\fp_zero:c
\fp_gzero:N
\fp_gzero:c

Setting a floating point to zero: copy `\c_zero_fp`.

```
\cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }
\cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }
\cs_generate_variant:Nn \fp_zero:N  { c }
\cs_generate_variant:Nn \fp_gzero:N { c }
```

(*End definition for* `\fp_zero:N` *and others. These functions are documented on page* **??**.)

\fp_zero_new:N
\fp_zero_new:c
\fp_gzero_new:N
\fp_gzero_new:c

Set the floating point to zero, or define it if needed.

```
\cs_new_protected:Npn \fp_zero_new:N #1
  { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
\cs_new_protected:Npn \fp_gzero_new:N #1
  { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
\cs_generate_variant:Nn \fp_zero_new:N  { c }
\cs_generate_variant:Nn \fp_gzero_new:N { c }
```

(*End definition for* `\fp_zero_new:N` *and others. These functions are documented on page* **??**.)

33.2 Updating values

These match the equivalent functions in l3int and l3skip.

\fp_add:Nn \fp_add:cn \fp_gadd:Nn \fp_gadd:cn \fp_sub:Nn \fp_sub:cn \fp_gsub:Nn \fp_gsub:cn __fp_add:NNNn

For the sake of error recovery we should not simply set `#1` to `#1` ± (`#2`): for instance, if `#2` is `0)+2`, the parsing error would be raised at the last closing parenthesis rather than at the closing parenthesis in the user argument. Thus we evaluate `#2` instead of just putting parentheses. As an optimization we use `\__fp_parse:n` rather than `\fp_eval:n`, which would convert the result away from the internal representation and back.

```
\cs_new_protected_nopar:Npn \fp_add:Nn  { \__fp_add:NNNn \fp_set:Nn  + }
\cs_new_protected_nopar:Npn \fp_gadd:Nn { \__fp_add:NNNn \fp_gset:Nn + }
\cs_new_protected_nopar:Npn \fp_sub:Nn  { \__fp_add:NNNn \fp_set:Nn  - }
\cs_new_protected_nopar:Npn \fp_gsub:Nn { \__fp_add:NNNn \fp_gset:Nn - }
\cs_new_protected:Npn \__fp_add:NNNn #1#2#3#4
  { #1 #3 { #3 #2 \__fp_parse:n {#4} } }
\cs_generate_variant:Nn \fp_add:Nn  { c }
\cs_generate_variant:Nn \fp_gadd:Nn { c }
\cs_generate_variant:Nn \fp_sub:Nn  { c }
\cs_generate_variant:Nn \fp_gsub:Nn { c }
```

(*End definition for* `\fp_add:Nn` *and others. These functions are documented on page* **??**.)

33.3 Showing values

\fp_show:N \fp_show:c \fp_show:n

This shows the result of computing its argument. The `\__msg_show_variable:n` auxiliary expects its input in a slightly odd form, starting with `>~`, and displays the rest.

```
\cs_new_protected:Npn \fp_show:N #1
  {
    \fp_if_exist:NTF #1
      { \__msg_show_variable:n { > ~ \fp_to_tl:N #1 } }
      {
        \__msg_kernel_error:nnx { kernel } { variable-not-defined }
          { \token_to_str:N #1 }
      }
  }
\cs_new_protected:Npn \fp_show:n #1
  { \__msg_show_variable:n { > ~ \fp_to_tl:n {#1} } }
\cs_generate_variant:Nn \fp_show:N { c }
```

(*End definition for* `\fp_show:N`*,* `\fp_show:c`*, and* `\fp_show:n`*. These functions are documented on page* **??**.)

33.4 Some useful constants and scratch variables

\c_one_fp \c_e_fp

Some constants.

```
\fp_const:Nn \c_e_fp          { 2.718 2818 2845 9045 }
\fp_const:Nn \c_one_fp        { 1 }
```

(*End definition for* `\c_one_fp` *and* `\c_e_fp`*. These variables are documented on page 176.*)

\c_pi_fp \c_one_degree_fp

We simply round π to the closest multiple of 10^{-15}.

```
\fp_const:Nn \c_pi_fp         { 3.141 5926 5358 9793 }
\fp_const:Nn \c_one_degree_fp { 0.0 1745 3292 5199 4330 }
```

(*End definition for* `\c_pi_fp` *and* `\c_one_degree_fp`*. These variables are documented on page 176.*)

\l_tmpa_fp
\l_tmpb_fp
\g_tmpa_fp
\g_tmpb_fp

Scratch variables are simply initialized there.

```
\fp_new:N \l_tmpa_fp
\fp_new:N \l_tmpb_fp
\fp_new:N \g_tmpa_fp
\fp_new:N \g_tmpb_fp
```

(*End definition for* `\l_tmpa_fp` *and others. These variables are documented on page 176.*)

```
⟨/initex | package⟩
```

34 l3fp-old implementation

```
⟨*initex | package⟩
⟨@@=fp⟩
```

34.1 Compatibility

\c_undefined_fp

The old floating point number `\c_undefined_fp` is now implemented as a `nan`.

```
\fp_const:Nn \c_undefined_fp { nan }
```

(*End definition for* `\c_undefined_fp`*. This variable is documented on page* ??.)

\fp_if_undefined_p:N
\fp_if_undefined:NTF

An old floating point is undefined if it is `inf` or `nan`, *i.e.*, if its type is 2 or 3.

```
\prg_new_conditional:Npnn \fp_if_undefined:N #1 { p , T , F , TF }
  { \exp_after:wN \__fp_if_undefined:w #1 }
\cs_new:Npn \__fp_if_undefined:w \s__fp \__fp_chk:w #1#2;
  {
    \if_int_compare:w #1 > \c_one
      \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\fp_if_undefined:N`*. These functions are documented on page* ??.)

\fp_if_zero_p:N
\fp_if_zero:NTF

An old floating point is zero if it is ± 0, *i.e.*, its type is 0.

```
\prg_new_conditional:Npnn \fp_if_zero:N #1 { p , T , F , TF }
  { \exp_after:wN \__fp_if_zero:w #1 }
\cs_new:Npn \__fp_if_zero:w \s__fp \__fp_chk:w #1#2;
  { \if_meaning:w #1 0 \prg_return_true: \else: \prg_return_false: \fi: }
```

(*End definition for* `\fp_if_zero:N`*. These functions are documented on page* ??.)

\fp_abs:N
\fp_abs:c
\fp_gabs:N
\fp_gabs:c
\fp_neg:N
\fp_neg:c
\fp_gneg:N
\fp_gneg:c
__fp_abs:NNN

Simply expand the floating point variable to feed it to `\__fp_abs_o:w` or `\__fp_-_o:w`, expanded within an expanding token list assignment. The `\prg_do_nothing:` is not necessary, but it reminds us more clearly that `\__fp_abs_o:w` and `\__fp_-_o:w` expand after their result.

```
\cs_new_protected_nopar:Npn \fp_abs:N  { \__fp_abs:NNN \tl_set:Nx  \__fp_abs_o:w }
\cs_new_protected_nopar:Npn \fp_gabs:N { \__fp_abs:NNN \tl_gset:Nx \__fp_abs_o:w }
\cs_new_protected_nopar:Npx \fp_neg:N
  {
    \exp_not:N \__fp_abs:NNN
    \exp_not:N \tl_set:Nx
    \exp_not:c { __fp_-_o:w }
```

```
  }
\cs_new_protected_nopar:Npx \fp_gneg:N
  {
    \exp_not:N \__fp_abs:NNN
    \exp_not:N \tl_gset:Nx
    \exp_not:c { __fp_-_o:w }
  }
\cs_new_protected:Npn \__fp_abs:NNN #1#2#3
  { #1 #3 { \exp_after:wN #2 #3 \prg_do_nothing: } }
\cs_generate_variant:Nn \fp_abs:N  { c }
\cs_generate_variant:Nn \fp_gabs:N { c }
\cs_generate_variant:Nn \fp_neg:N  { c }
\cs_generate_variant:Nn \fp_gneg:N { c }
```

(*End definition for* `\fp_abs:N` *and others. These functions are documented on page* **??**.)

`\fp_mul:Nn` `\fp_mul:cn` `\fp_gmul:Nn` `\fp_gmul:cn` `\fp_div:Nn` `\fp_div:cn` `\fp_gdiv:Nn` `\fp_gdiv:cn` `\fp_pow:Nn` `\fp_pow:cn` `\fp_gpow:Nn` `\fp_gpow:cn` `\__fp_mul:NNNn`

See `\fp_add:Nn` for details.

```
\cs_new_protected_nopar:Npn \fp_mul:Nn  { \__fp_mul:NNNn \fp_set:Nn  * }
\cs_new_protected_nopar:Npn \fp_gmul:Nn { \__fp_mul:NNNn \fp_gset:Nn * }
\cs_new_protected_nopar:Npn \fp_div:Nn  { \__fp_mul:NNNn \fp_set:Nn  / }
\cs_new_protected_nopar:Npn \fp_gdiv:Nn { \__fp_mul:NNNn \fp_gset:Nn / }
\cs_new_protected_nopar:Npn \fp_pow:Nn  { \__fp_mul:NNNn \fp_set:Nn  ^ }
\cs_new_protected_nopar:Npn \fp_gpow:Nn { \__fp_mul:NNNn \fp_gset:Nn ^ }
\cs_new_protected:Npn \__fp_mul:NNNn #1#2#3#4
  { #1 #3 { #3 #2 \__fp_parse:n {#4} } }
\cs_generate_variant:Nn \fp_mul:Nn  { c }
\cs_generate_variant:Nn \fp_gmul:Nn { c }
\cs_generate_variant:Nn \fp_div:Nn  { c }
\cs_generate_variant:Nn \fp_gdiv:Nn { c }
\cs_generate_variant:Nn \fp_pow:Nn  { c }
\cs_generate_variant:Nn \fp_gpow:Nn { c }
```

(*End definition for* `\fp_mul:Nn` *and others. These functions are documented on page* **??**.)

`\fp_exp:Nn` `\fp_exp:cn` `\fp_gexp:Nn` `\fp_gexp:cn` `\fp_ln:Nn` `\fp_ln:cn` `\fp_gln:Nn` `\fp_gln:cn` `\fp_sin:Nn` `\fp_sin:cn` `\fp_gsin:Nn` `\fp_gsin:cn` `\fp_cos:Nn` `\fp_cos:cn` `\fp_gcos:Nn` `\fp_gcos:cn` `\fp_tan:Nn` `\fp_tan:cn` `\fp_gtan:Nn` `\fp_gtan:cn` `\__fp_assign_to:nNNNn` `\__fp_assign_to_i:wNNNn` `\__fp_assign_to_ii:NnNNN`

Here, an added twist is that each value computed by these expensive unary operations is stored as a constant floating point number.

```
\cs_set_protected:Npn \__fp_tmp:w #1#2#3#4#5
  {
    \cs_new_protected_nopar:Npn #1 { #5 {#4} \tl_set_eq:NN  #3 }
    \cs_new_protected_nopar:Npn #2 { #5 {#4} \tl_gset_eq:NN #3 }
    \cs_generate_variant:Nn #1 { c }
    \cs_generate_variant:Nn #2 { c }
  }
\__fp_tmp:w \fp_exp:Nn \fp_gexp:Nn \__fp_exp_o:w {exp} \__fp_assign_to:nNNNn
\__fp_tmp:w \fp_ln:Nn  \fp_gln:Nn  \__fp_ln_o:w  {ln } \__fp_assign_to:nNNNn
\__fp_tmp:w \fp_sin:Nn \fp_gsin:Nn \__fp_sin_o:w {sin} \__fp_assign_to:nNNNn
\__fp_tmp:w \fp_cos:Nn \fp_gcos:Nn \__fp_cos_o:w {cos} \__fp_assign_to:nNNNn
\__fp_tmp:w \fp_tan:Nn \fp_gtan:Nn \__fp_tan_o:w {tan} \__fp_assign_to:nNNNn
\cs_new_protected:Npn \__fp_assign_to:nNNNn #1#2#3#4#5
  {
    \exp_after:wN \__fp_assign_to_i:wNNNn
```

```
    \tex_romannumeral:D -`0 \__fp_parse:n {#5} {#1} #2#3#4
  }
\cs_new_protected:Npn \__fp_assign_to_i:wNNNn \s__fp \__fp_chk:w #1#2#3; #4
  {
    \exp_args:Nc \__fp_assign_to_ii:NnNNN
      { c__fp_ #4 [ #1 # 2 \if_meaning:w 1 #1 #3 \fi: ] _fp }
      { #1#2#3 }
  }
\cs_new_protected:Npn \__fp_assign_to_ii:NnNNN #1#2#3#4#5
  {
    \cs_if_exist:NF #1
      { \tl_const:Nx #1 { #4 \s__fp \__fp_chk:w #2; } }
    #3 #5 #1
  }
```

(*End definition for* `\fp_exp:Nn` *and others. These functions are documented on page* **??**.)

`\fp_compare:NNNTF`

Comparisons used to be easier between floating points stored in variables. No more.

```
\cs_new_protected_nopar:Npn \fp_compare:NNNTF { \fp_compare:nNnTF }
\cs_new_protected_nopar:Npn \fp_compare:NNNT  { \fp_compare:nNnT  }
\cs_new_protected_nopar:Npn \fp_compare:NNNF  { \fp_compare:nNnF  }
```

(*End definition for* `\fp_compare:NNNTF`*. This function is documented on page* **??**.)

`\fp_round_places:Nn`
`\fp_ground_places:Nn`
`\__fp_round_places:NNn`

Rounding to a given number of places is easy, since it is provided by the l3fp-round module.

```
\cs_new_protected_nopar:Npn \fp_round_places:Nn
  { \__fp_round_places:NNn \tl_set:Nx }
\cs_new_protected_nopar:Npn \fp_ground_places:Nn
  { \__fp_round_places:NNn \tl_gset:Nx }
\cs_new_protected:Npn \__fp_round_places:NNn #1#2#3
  {
    #1 #2
      {
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_round:Nwn
        \exp_after:wN \exp_after:wN
        \exp_after:wN \__fp_round_to_nearest:NNN
        \exp_after:wN #2
        \exp_after:wN { \int_use:N \__int_eval:w #3 }
      }
  }
\cs_generate_variant:Nn \fp_round_places:Nn { c }
\cs_generate_variant:Nn \fp_ground_places:Nn { c }
```

(*End definition for* `\fp_round_places:Nn` *and* `\fp_ground_places:Nn`*. These functions are documented on page* **??**.)

`\fp_round_figures:Nn`
`\fp_ground_figures:Nn`

Rounding to a given number of figures is the same as rounding to a number of places, after shifting by the exponent of the argument.

```
\cs_new_protected:Npn \fp_round_figures:Nn #1#2
  {
```

```
    \__fp_round_places:NNn \tl_set:Nx #1
      { #2 - \exp_after:wN \__fp_exponent:w #1 }
  }
\cs_new_protected:Npn \fp_ground_figures:Nn #1#2
  {
    \__fp_round_places:NNn \tl_gset:Nx #1
      { #2 - \exp_after:wN \__fp_exponent:w #1 }
  }
\cs_generate_variant:Nn \fp_round_figures:Nn { c }
\cs_generate_variant:Nn \fp_ground_figures:Nn { c }
```

(*End definition for* `\fp_round_figures:Nn` *and* `\fp_ground_figures:Nn`*. These functions are documented on page* **??**.)

```
⟨/initex | package⟩
```

35 l3luatex implementation

```
⟨*initex | package⟩
```

`\lua_now_x:n`
`\lua_now_x:x`
`\lua_now:n`
`\lua_now:x`
`\lua_shipout_x:n`
`\lua_shipout_x:x`
`\lua_shipout:n`
`\lua_shipout:x`

When LuaTEX is in use, this is all a question of primitives with new names. On the other hand, for pdfTEX and XƎTEX the argument should be removed from the input stream before issuing an error. This is expandable, using `\__msg_kernel_expandable_error:nnn` as done for V-type expansion in l3expan.

```
\luatex_if_engine:TF
  {
    \cs_new_eq:NN \lua_now_x:n     \luatex_directlua:D
    \cs_new_eq:NN \lua_shipout_x:n \luatex_latelua:D
  }
  {
    \cs_new:Npn \lua_now_x:n #1
      {
        \__msg_kernel_expandable_error:nnn
          { kernel } { bad-engine } { \lua_now_x:n }
      }
    \cs_new_protected:Npn \lua_shipout_x:n #1
      {
        \__msg_kernel_expandable_error:nnn
          { kernel } { bad-engine } { \lua_shipout_x:n }
      }
  }
\cs_generate_variant:Nn \lua_now_x:n { x }
\cs_new:Npn \lua_now:n #1
  { \lua_now_x:n { \exp_not:n {#1} } }
\cs_generate_variant:Nn \lua_now:n { x }
\cs_generate_variant:Nn \lua_shipout_x:n { x }
\cs_new_protected:Npn \lua_shipout:n #1
  { \lua_shipout_x:n { \exp_not:n {#1} } }
\cs_generate_variant:Nn \lua_shipout:n { x }
```

(*End definition for* `\lua_now_x:n` *and* `\lua_now_x:x`*. These functions are documented on page* **??**.)

35.1 Category code tables

```
⟨@@=cctab⟩
```

\g__cctab_allocate_int
\g__cctab_stack_int
\g__cctab_stack_seq

To allocate category code tables, both the read-only and stack tables need to be followed. There is also a sequence stack for the dynamic tables themselves.

```
\int_new:N  \g__cctab_allocate_int
\int_set:Nn \g__cctab_allocate_int { \c_minus_one }
\int_new:N \g__cctab_stack_int
\seq_new:N \g__cctab_stack_seq
```

(*End definition for* \g__cctab_allocate_int*. This variable is documented on page* **??**.)

\cctab_new:N

Creating a new category code table is done slightly differently from other registers. Low-numbered tables are more efficiently-stored than high-numbered ones. There is also a need to have a stack of flexible tables as well as the set of read-only ones. To satisfy both of these requirements, odd numbered tables are used for read-only tables, and even ones for the stack. Here, therefore, the odd numbers are allocated.

```
\cs_new_protected:Npn \cctab_new:N #1
  {
    \__chk_if_free_cs:N #1
    \int_gadd:Nn \g__cctab_allocate_int { \c_two }
    \int_compare:nNnTF
      \g__cctab_allocate_int <  { \c_max_register_int + \c_one }
       {
         \tex_global:D \tex_chardef:D #1 \g__cctab_allocate_int
         \luatex_initcatcodetable:D #1
       }
       { \__msg_kernel_fatal:nnx { kernel } { out-of-registers } { cctab } }
  }
\luatex_if_engine:F
  {
    \cs_set_protected:Npn \cctab_new:N #1
      {
        \__msg_kernel_error:nnx { kernel } { bad-engine }
          { \exp_not:N \cctab_new:N }
      }
  }
⟨*package⟩
\luatex_if_engine:T
  {
    \cs_set_protected:Npn \cctab_new:N #1
      {
        \__chk_if_free_cs:N #1
        \newcatcodetable #1
        \luatex_initcatcodetable:D #1
      }
  }
⟨/package⟩
```

(*End definition for* \cctab_new:N*. This function is documented on page* 190.)

\cctab_begin:N
\cctab_end:
\l__cctab_internal_tl

The aim here is to ensure that the saved tables are read-only. This is done by using a stack of tables which are not read only, and actually having them as “in use” copies.

```
\cs_new_protected:Npn \cctab_begin:N #1
  {
    \seq_gpush:Nx \g__cctab_stack_seq { \tex_the:D \luatex_catcodetable:D }
    \luatex_catcodetable:D #1
    \int_gadd:Nn \g__cctab_stack_int { \c_two }
    \int_compare:nNnT \g__cctab_stack_int > \c_max_register_int
      { \__msg_kernel_fatal:nn { kernel } { cctab-stack-full } }
    \luatex_savecatcodetable:D \g__cctab_stack_int
    \luatex_catcodetable:D \g__cctab_stack_int
  }
\cs_new_protected_nopar:Npn \cctab_end:
  {
    \int_gsub:Nn \g__cctab_stack_int { \c_two }
    \seq_if_empty:NTF \g__cctab_stack_seq
      { \tl_set:Nn \l__cctab_internal_tl { 0 } }
      { \seq_gpop:NN \g__cctab_stack_seq \l__cctab_internal_tl }
    \luatex_catcodetable:D \l__cctab_internal_tl \scan_stop:
  }
\luatex_if_engine:F
  {
    \cs_set_protected:Npn \cctab_begin:N #1
      {
        \__msg_kernel_error:nnxx { kernel } { bad-engine }
          { \exp_not:N \cctab_begin:N } {#1}
      }
    \cs_set_protected_nopar:Npn \cctab_end:
      {
        \__msg_kernel_error:nnx { kernel } { bad-engine }
          { \exp_not:N \cctab_end: }
      }
  }
⟨*package⟩
\luatex_if_engine:T
  {
    \cs_set_protected:Npn \cctab_begin:N #1 { \BeginCatcodeRegime #1 }
    \cs_set_protected_nopar:Npn \cctab_end: { \EndCatcodeRegime }
  }
⟨/package⟩
\tl_new:N \l__cctab_internal_tl
```

(*End definition for* `\cctab_begin:N`*. This function is documented on page* **??**.)

\cctab_gset:Nn

Category code tables are always global, so only one version is needed. The set up here is simple, and means that at the point of use there is no need to worry about escaping category codes.

```
\cs_new_protected:Npn \cctab_gset:Nn #1#2
  {
    \group_begin:
```

```
      #2
      \luatex_savecatcodetable:D #1
    \group_end:
  }
\luatex_if_engine:F
  {
    \cs_set_protected:Npn \cctab_gset:Nn #1#2
      {
        \__msg_kernel_error:nnxx { kernel } { bad-engine }
          { \exp_not:N \cctab_gset:Nn } { #1 {#2} }
      }
  }
```

(*End definition for* `\cctab_gset:Nn`*. This function is documented on page 190.*)

`\c_code_cctab`
`\c_document_cctab`
`\c_initex_cctab`
`\c_other_cctab`
`\c_str_cctab`

Creating category code tables is easy using the function above. The `other` and `string` ones are done by completely ignoring the existing codes as this makes life a lot less complex. The table for expl3 category codes is always needed, whereas when in package mode the rest can be copied from the existing $\LaTeX\,2_\varepsilon$ package luatex.

```
\luatex_if_engine:T
  {
    \cctab_new:N \c_code_cctab
    \cctab_gset:Nn \c_code_cctab { }
  }
⟨*package⟩
\luatex_if_engine:T
  {
    \cs_new_eq:NN \c_document_cctab \CatcodeTableLaTeX
    \cs_new_eq:NN \c_initex_cctab   \CatcodeTableIniTeX
    \cs_new_eq:NN \c_other_cctab    \CatcodeTableOther
    \cs_new_eq:NN \c_str_cctab      \CatcodeTableString
  }
⟨/package⟩
⟨*initex⟩
\luatex_if_engine:T
  {
    \cctab_new:N \c_document_cctab
    \cctab_new:N \c_other_cctab
    \cctab_new:N \c_str_cctab
    \cctab_gset:Nn \c_document_cctab
      {
        \char_set_catcode_space:n          { 9 }
        \char_set_catcode_space:n          { 32 }
        \char_set_catcode_other:n          { 58 }
        \char_set_catcode_math_subscript:n { 95 }
        \char_set_catcode_active:n         { 126 }
      }
    \cctab_gset:Nn \c_other_cctab
      {
        \int_step_inline:nnnn { 0 } { 1 } { 127 }
```

```
          { \char_set_catcode_other:n {#1} }
      }
    \cctab_gset:Nn \c_str_cctab
      {
        \int_step_inline:nnnn { 0 } { 1 } { 127 }
          { \char_set_catcode_other:n {#1} }
        \char_set_catcode_space:n { 32 }
      }
  }
⟨/initex⟩
```

(*End definition for* `\c_code_cctab`. *This variable is documented on page 191.*)

35.2 Messages

```
\__msg_kernel_new:nnnn { kernel } { bad-engine }
  { LuaTeX~engine~not~in~use!~Ignoring~#1. }
  {
    The~feature~you~are~using~is~only~available~
    with~the~LuaTeX~engine.~LaTeX3~ignored~'#1#2'.
  }
\__msg_kernel_new:nnnn { kernel } { cctab-stack-full }
  { The~category~code~table~stack~is~exhausted. }
  {
    LaTeX~has~been~asked~to~switch~to~a~new~category~code~table,~
    but~there~is~no~more~space~to~do~this!
  }
```

```
⟨/initex | package⟩
```

36 l3candidates Implementation

```
⟨*initex | package⟩
```

36.1 Additions to l3box

```
⟨@@=box⟩
```

36.2 Affine transformations

`\l__box_angle_fp`

When rotating boxes, the angle itself may be needed by the engine-dependent code. This is done using the `fp` module so that the value is tidied up properly.

```
\fp_new:N \l__box_angle_fp
```

(*End definition for* `\l__box_angle_fp`. *This variable is documented on page 194.*)

`\l__box_cos_fp`
`\l__box_sin_fp`

These are used to hold the calculated sine and cosine values while carrying out a rotation.

```
\fp_new:N \l__box_cos_fp
\fp_new:N \l__box_sin_fp
```

(*End definition for* `\l__box_cos_fp` *and* `\l__box_sin_fp`. *These variables are documented on page 194.*)

\l__box_top_dim
\l__box_bottom_dim
\l__box_left_dim
\l__box_right_dim

These are the positions of the four edges of a box before manipulation.

```
\dim_new:N \l__box_top_dim
\dim_new:N \l__box_bottom_dim
\dim_new:N \l__box_left_dim
\dim_new:N \l__box_right_dim
```

(*End definition for* `\l__box_top_dim` *and others. These variables are documented on page* **??**.)

\l__box_top_new_dim
\l__box_bottom_new_dim
\l__box_left_new_dim
\l__box_right_new_dim

These are the positions of the four edges of a box after manipulation.

```
\dim_new:N \l__box_top_new_dim
\dim_new:N \l__box_bottom_new_dim
\dim_new:N \l__box_left_new_dim
\dim_new:N \l__box_right_new_dim
```

(*End definition for* `\l__box_top_new_dim` *and others. These variables are documented on page* **??**.)

\l__box_internal_box

Scratch space, but also needed by some parts of the driver.

```
\box_new:N \l__box_internal_box
```

(*End definition for* `\l__box_internal_box`*. This variable is documented on page 194.*)

\box_rotate:Nn
__box_rotate:N
__box_rotate_x:nnN
__box_rotate_y:nnN
__box_rotate_quadrant_one:
__box_rotate_quadrant_two:
__box_rotate_quadrant_three:
__box_rotate_quadrant_four:

Rotation of a box starts with working out the relevant sine and cosine. The actual rotation is in an auxiliary to keep the flow slightly clearer

```
\cs_new_protected:Npn \box_rotate:Nn #1#2
  {
    \hbox_set:Nn #1
      {
        \group_begin:
          \fp_set:Nn \l__box_angle_fp {#2}
          \fp_set:Nn \l__box_sin_fp { sind ( \l__box_angle_fp ) }
          \fp_set:Nn \l__box_cos_fp { cosd ( \l__box_angle_fp ) }
          \__box_rotate:N #1
        \group_end:
      }
  }
```

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.

```
\cs_new_protected:Npn \__box_rotate:N #1
  {
    \dim_set:Nn \l__box_top_dim    {  \box_ht:N #1 }
    \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
    \dim_set:Nn \l__box_right_dim  {  \box_wd:N #1 }
    \dim_zero:N \l__box_left_dim
```

The next step is to work out the x and y coordinates of vertices of the rotated box in relation to its original coordinates. The box can be visualized with vertices B, C, D and E is illustrated (Figure 1). The vertex O is the reference point on the baseline, and in this implementation is also the centre of rotation. The formulae are, for a point P and

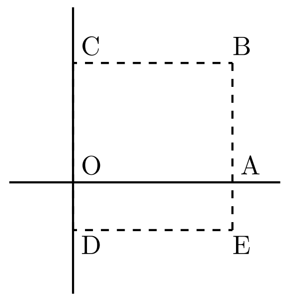

Figure 1: Co-ordinates of a box prior to rotation.

angle α:

$$
\begin{aligned}
P'_x &= P_x - O_x \\
P'_y &= P_y - O_y \\
P''_x &= (P'_x \cos(\alpha)) - (P'_y \sin(\alpha)) \\
P''_y &= (P'_x \sin(\alpha)) + (P'_y \cos(\alpha)) \\
P'''_x &= P''_x + O_x + L_x \\
P'''_y &= P''_y + O_y
\end{aligned}
$$

The "extra" horizontal translation L_x at the end is calculated so that the leftmost point of the resulting box has x-coordinate 0. This is desirable as TEX boxes must have the reference point at the left edge of the box. (As O is always $(0,0)$, this part of the calculation is omitted here.)

```
    \fp_compare:nNnTF \l__box_sin_fp > \c_zero_fp
      {
        \fp_compare:nNnTF \l__box_cos_fp > \c_zero_fp
          { \__box_rotate_quadrant_one: }
          { \__box_rotate_quadrant_two: }
      }
      {
        \fp_compare:nNnTF \l__box_cos_fp < \c_zero_fp
          { \__box_rotate_quadrant_three: }
          { \__box_rotate_quadrant_four: }
      }
```

The position of the box edges are now known, but the box at this stage be misplaced relative to the current TEX reference point. So the content of the box is moved such that the reference point of the rotated box will be in the same place as the original.

```
    \hbox_set:Nn \l__box_internal_box { \box_use:N #1 }
    \hbox_set:Nn \l__box_internal_box
      {
        \tex_kern:D -\l__box_left_new_dim
        \hbox:n
          {
            \__driver_box_rotate_begin:
            \box_use:N \l__box_internal_box
            \__driver_box_rotate_end:
```

```
          }
      }
```

Tidy up the size of the box so that the material is actually inside the bounding box. The result can then be used to reset the original box.

```
    \box_set_ht:Nn \l__box_internal_box {  \l__box_top_new_dim }
    \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
    \box_set_wd:Nn \l__box_internal_box
      { \l__box_right_new_dim - \l__box_left_new_dim }
    \box_use:N \l__box_internal_box
  }
```

These functions take a general point (#1, #2) and rotate its location about the origin, using the previously-set sine and cosine values. Each function gives only one component of the location of the updated point. This is because for rotation of a box each step needs only one value, and so performance is gained by avoiding working out both x' and y' at the same time. Contrast this with the equivalent function in the l3coffins module, where both parts are needed.

```
\cs_new_protected:Npn \__box_rotate_x:nnN #1#2#3
  {
    \dim_set:Nn #3
      {
        \fp_to_dim:n
          {
                \l__box_cos_fp * \dim_to_fp:n {#1}
            - ( \l__box_sin_fp * \dim_to_fp:n {#2} )
          }
      }
  }
\cs_new_protected:Npn \__box_rotate_y:nnN #1#2#3
  {
    \dim_set:Nn #3
      {
        \fp_to_dim:n
          {
              \l__box_sin_fp * \dim_to_fp:n {#1}
            + \l__box_cos_fp * \dim_to_fp:n {#2}
          }
      }
  }
```

Rotation of the edges is done using a different formula for each quadrant. In every case, the top and bottom edges only need the resulting y-values, whereas the left and right edges need the x-values. Each case is a question of picking out which corner ends up at with the maximum top, bottom, left and right value. Doing this by hand means a lot less calculating and avoids lots of comparisons.

```
\cs_new_protected:Npn \__box_rotate_quadrant_one:
  {
    \__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
      \l__box_top_new_dim
```

```
    \__box_rotate_y:nnN \l__box_left_dim  \l__box_bottom_dim
      \l__box_bottom_new_dim
    \__box_rotate_x:nnN \l__box_left_dim  \l__box_top_dim
      \l__box_left_new_dim
    \__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
      \l__box_right_new_dim
  }
\cs_new_protected:Npn \__box_rotate_quadrant_two:
  {
    \__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
      \l__box_top_new_dim
    \__box_rotate_y:nnN \l__box_left_dim  \l__box_top_dim
      \l__box_bottom_new_dim
    \__box_rotate_x:nnN \l__box_right_dim  \l__box_top_dim
      \l__box_left_new_dim
    \__box_rotate_x:nnN \l__box_left_dim   \l__box_bottom_dim
      \l__box_right_new_dim
  }
\cs_new_protected:Npn \__box_rotate_quadrant_three:
  {
    \__box_rotate_y:nnN \l__box_left_dim  \l__box_bottom_dim
      \l__box_top_new_dim
    \__box_rotate_y:nnN \l__box_right_dim \l__box_top_dim
      \l__box_bottom_new_dim
    \__box_rotate_x:nnN \l__box_right_dim \l__box_bottom_dim
      \l__box_left_new_dim
    \__box_rotate_x:nnN \l__box_left_dim   \l__box_top_dim
      \l__box_right_new_dim
  }
\cs_new_protected:Npn \__box_rotate_quadrant_four:
  {
    \__box_rotate_y:nnN \l__box_left_dim  \l__box_top_dim
      \l__box_top_new_dim
    \__box_rotate_y:nnN \l__box_right_dim \l__box_bottom_dim
      \l__box_bottom_new_dim
    \__box_rotate_x:nnN \l__box_left_dim  \l__box_bottom_dim
      \l__box_left_new_dim
    \__box_rotate_x:nnN \l__box_right_dim \l__box_top_dim
      \l__box_right_new_dim
  }
```

(*End definition for* `\box_rotate:Nn`*. This function is documented on page* *193*.)

`\l__box_scale_x_fp`
`\l__box_scale_y_fp`

Scaling is potentially-different in the two axes.

```
\fp_new:N \l__box_scale_x_fp
\fp_new:N \l__box_scale_y_fp
```

(*End definition for* `\l__box_scale_x_fp` *and* `\l__box_scale_y_fp`*. These variables are documented on page* *194*.)

`\box_resize:Nnn`
`\box_resize:cnn`
`\__box_resize:Nnn`

Resizing a box starts by working out the various dimensions of the existing box.

```
\cs_new_protected:Npn \box_resize:Nnn #1#2#3
  {
    \hbox_set:Nn #1
      {
        \group_begin:
          \dim_set:Nn \l__box_top_dim    {  \box_ht:N #1 }
          \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
          \dim_set:Nn \l__box_right_dim  {  \box_wd:N #1 }
          \dim_zero:N \l__box_left_dim
```

The x-scaling and resulting box size is easy enough to work out: the dimension is that given as #2, and the scale is simply the new width divided by the old one.

```
          \fp_set:Nn \l__box_scale_x_fp
            { \dim_to_fp:n {#2} / ( \dim_to_fp:n \l__box_right_dim ) }
```

The y-scaling needs both the height and the depth of the current box.

```
          \fp_set:Nn \l__box_scale_y_fp
            {
              \dim_to_fp:n {#3} /
                ( \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim } )
            }
```

Hand off to the auxiliary which does the work.

```
          \__box_resize:Nnn #1 {#2} {#3}
        \group_end:
      }
  }
\cs_generate_variant:Nn \box_resize:Nnn { c }
```

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. This is done using the absolute value of the scale so that the new edge is in the correct place. In the y direction, both dimensions have to be scaled, and this again needs the absolute scale value. Once that is all done, the common resize/rescale code can be employed.

```
\cs_new_protected:Npn \__box_resize:Nnn #1#2#3
  {
    \dim_set:Nn \l__box_right_new_dim { \dim_abs:n {#2} }
    \dim_set:Nn \l__box_bottom_new_dim
      { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
    \dim_set:Nn \l__box_top_new_dim
      { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
    \__box_resize_common:N #1
  }
```

(*End definition for* `\box_resize:Nnn` *and* `\box_resize:cnn`*. These functions are documented on page* ??*.*)

\box_resize_to_ht_plus_dp:Nn
\box_resize_to_ht_plus_dp:cn
\box_resize_to_wd:Nn
\box_resize_to_wd:cn

Scaling to a total height or to a width is a simplified version of the main resizing operation, with the scale simply copied between the two parts. The internal auxiliary is called using the scaling value twice, as the sign for both parts is needed (as this allows the same internal code to be used as for the general case).

```
\cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2
  {
    \hbox_set:Nn #1
      {
        \group_begin:
          \dim_set:Nn \l__box_top_dim    {  \box_ht:N #1 }
          \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
          \dim_set:Nn \l__box_right_dim  {  \box_wd:N #1 }
          \dim_zero:N \l__box_left_dim
          \fp_set:Nn \l__box_scale_y_fp
            {
              \dim_to_fp:n {#2} /
                ( \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim } )
            }
          \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
          \__box_resize:Nnn #1 {#2} {#2}
        \group_end:
      }
  }
\cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
\cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
  {
    \hbox_set:Nn #1
      {
        \group_begin:
          \dim_set:Nn \l__box_top_dim    {  \box_ht:N #1 }
          \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
          \dim_set:Nn \l__box_right_dim  {  \box_wd:N #1 }
          \dim_zero:N \l__box_left_dim
          \fp_set:Nn \l__box_scale_x_fp
            { \dim_to_fp:n {#2} / ( \dim_to_fp:n \l__box_right_dim ) }
          \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
          \__box_resize:Nnn #1 {#2} {#2}
        \group_end:
      }
  }
\cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
```

(*End definition for* `\box_resize_to_ht_plus_dp:Nn` *and* `\box_resize_to_ht_plus_dp:cn`*. These functions are documented on page* **??***.*)

`\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.

```
\cs_new_protected:Npn \box_scale:Nnn #1#2#3
  {
    \hbox_set:Nn #1
      {
```

```
        \group_begin:
          \fp_set:Nn \l__box_scale_x_fp {#2}
          \fp_set:Nn \l__box_scale_y_fp {#3}
          \dim_set:Nn \l__box_top_dim    {  \box_ht:N #1 }
          \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
          \dim_set:Nn \l__box_right_dim  {  \box_wd:N #1 }
          \dim_zero:N \l__box_left_dim
          \dim_set:Nn \l__box_top_new_dim
            { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
          \dim_set:Nn \l__box_bottom_new_dim
            { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
          \dim_set:Nn \l__box_right_new_dim
              { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
           \__box_resize_common:N #1
        \group_end:
      }
  }
\cs_generate_variant:Nn \box_scale:Nnn { c }
```

(*End definition for* `\box_scale:Nnn` *and* `\box_scale:cnn`*. These functions are documented on page* **??**.)

`\__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.

```
\cs_new_protected:Npn \__box_resize_common:N #1
  {
    \hbox_set:Nn \l__box_internal_box
      {
        \__driver_box_scale_begin:
        \hbox_overlap_right:n { \box_use:N #1 }
        \__driver_box_scale_end:
      }
```

The new height and depth can be applied directly.

```
    \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
    \box_set_dp:Nn \l__box_internal_box { \l__box_bottom_new_dim }
```

Things are not quite as obvious for the width, as the reference point needs to remain unchanged. For positive scaling factors resizing the box is all that is needed. However, for case of a negative scaling the material must be shifted such that the reference point ends up in the right place.

```
    \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
      {
        \hbox_to_wd:nn { \l__box_right_new_dim }
          {
            \tex_kern:D \l__box_right_new_dim
            \box_use:N \l__box_internal_box
            \tex_hss:D
          }
      }
      {
        \box_set_wd:Nn \l__box_internal_box { \l__box_right_new_dim }
```

```
        \hbox:n
          {
            \tex_kern:D \c_zero_dim
            \box_use:N \l__box_internal_box
            \tex_hss:D
          }
      }
  }
```

(*End definition for* `\__box_resize_common:N`*.*)

36.3 Viewing part of a box

`\box_clip:N`
`\box_clip:c`

A wrapper around the driver-dependent code.

```
\cs_new_protected:Npn \box_clip:N #1
  { \hbox_set:Nn #1 { \__driver_box_use_clip:N #1 } }
\cs_generate_variant:Nn \box_clip:N { c }
```

(*End definition for* `\box_clip:N` *and* `\box_clip:c`*. These functions are documented on page* **??***.*)

`\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.

```
\cs_new_protected:Npn \box_trim:Nnnnn #1#2#3#4#5
  {
    \hbox_set:Nn \l__box_internal_box
      {
        \tex_kern:D -\__dim_eval:w #2 \__dim_eval_end:
        \box_use:N #1
        \tex_kern:D -\__dim_eval:w #4 \__dim_eval_end:
      }
```

For the height and depth, there is a need to watch the baseline is respected. Material always has to stay on the correct side, so trimming has to check that there is enough material to trim. First, the bottom edge. If there is enough depth, simply set the depth, or if not move down so the result is zero depth. `\box_move_down:nn` is used in both cases so the resulting box always contains a `\lower` primitive. The internal box is used here as it allows safe use of `\box_set_dp:Nn`.

```
    \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_down:nn \c_zero_dim
              { \box_use:N \l__box_internal_box }
          }
        \box_set_dp:Nn \l__box_internal_box { \box_dp:N #1 - (#3) }
      }
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_down:nn { #3 - \box_dp:N #1 }
              { \box_use:N \l__box_internal_box }
```

```
        }
      \box_set_dp:Nn \l__box_internal_box \c_zero_dim
    }
```

Same thing, this time from the top of the box.

```
    \dim_compare:nNnTF { \box_ht:N \l__box_internal_box } > {#5}
      {
        \hbox_set:Nn \l__box_internal_box
          { \box_move_up:nn \c_zero_dim { \box_use:N \l__box_internal_box } }
        \box_set_ht:Nn \l__box_internal_box
          { \box_ht:N \l__box_internal_box - (#5) }
      }
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_up:nn { #5 - \box_ht:N \l__box_internal_box }
              { \box_use:N \l__box_internal_box }
          }
        \box_set_ht:Nn \l__box_internal_box \c_zero_dim
      }
    \box_set_eq:NN #1 \l__box_internal_box
  }
\cs_generate_variant:Nn \box_trim:Nnnnn { c }
```

(*End definition for* `\box_trim:Nnnnn` *and* `\box_trim:cnnnn`*. These functions are documented on page* ??.)

\box_viewport:Nnnnn
\box_viewport:cnnnn

The same general logic as for the trim operation, but with absolute dimensions. As a result, there are some things to watch out for in the vertical direction.

```
\cs_new_protected:Npn \box_viewport:Nnnnn #1#2#3#4#5
  {
    \hbox_set:Nn \l__box_internal_box
      {
        \tex_kern:D -\__dim_eval:w #2 \__dim_eval_end:
        \box_use:N #1
        \tex_kern:D \__dim_eval:w #4 - \box_wd:N #1 \__dim_eval_end:
      }
    \dim_compare:nNnTF {#3} < \c_zero_dim
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_down:nn \c_zero_dim
              { \box_use:N \l__box_internal_box }
          }
        \box_set_dp:Nn \l__box_internal_box { -\dim_eval:n {#3} }
      }
      {
        \hbox_set:Nn \l__box_internal_box
          { \box_move_down:nn {#3} { \box_use:N \l__box_internal_box } }
        \box_set_dp:Nn \l__box_internal_box \c_zero_dim
      }
```

```
    \dim_compare:nNnTF {#5} > \c_zero_dim
      {
        \hbox_set:Nn \l__box_internal_box
          { \box_move_up:nn \c_zero_dim { \box_use:N \l__box_internal_box } }
        \box_set_ht:Nn \l__box_internal_box
          {
            #5
            \dim_compare:nNnT {#3} > \c_zero_dim
              { - (#3) }
          }
      }
      {
        \hbox_set:Nn \l__box_internal_box
          {
            \box_move_up:nn { -\dim_eval:n {#5} }
              { \box_use:N \l__box_internal_box }
          }
        \box_set_ht:Nn \l__box_internal_box \c_zero_dim
      }
    \box_set_eq:NN #1 \l__box_internal_box
  }
\cs_generate_variant:Nn \box_viewport:Nnnnn { c }
```

(*End definition for* `\box_viewport:Nnnnn` *and* `\box_viewport:cnnnn`*. These functions are documented on page* ??.)

36.4 Additions to l3clist

```
⟨@@=clist⟩
```

`\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.

```
\cs_new:Npn \clist_item:Nn #1#2
  {
    \exp_args:Nfo \__clist_item:nnNn
      { \clist_count:N #1 }
      #1
      \__clist_item_N_loop:nw
      {#2}
  }
\cs_new:Npn \__clist_item:nnNn #1#2#3#4
  {
    \int_compare:nNnTF {#4} < \c_zero
      {
        \int_compare:nNnTF {#4} < { - #1 }
          { \use_none_delimit_by_q_stop:w }
          { \exp_args:Nf #3 { \int_eval:n { #4 + \c_one + #1 } } }
```

```
      }
      {
        \int_compare:nNnTF {#4} > {#1}
          { \use_none_delimit_by_q_stop:w }
          { #3 {#4} }
      }
    { } , #2 , \q_stop
  }
\cs_new:Npn \__clist_item_N_loop:nw #1 #2,
  {
    \int_compare:nNnTF {#1} = \c_zero
      { \use_i_delimit_by_q_stop:nw { \exp_not:n {#2} } }
      { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
  }
\cs_generate_variant:Nn \clist_item:Nn { c }
```

(*End definition for* `\clist_item:Nn` *and* `\clist_item:cn`*. These functions are documented on page* **??**.)

`\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.

```
\cs_new:Npn \clist_item:nn #1#2
  {
    \exp_args:Nf \__clist_item:nnNn
      { \clist_count:n {#1} }
      {#1}
      \__clist_item_n:nw
      {#2}
  }
\cs_new:Npn \__clist_item_n:nw #1
  { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
\cs_new:Npn \__clist_item_n_loop:nw #1 #2,
  {
    \exp_args:No \tl_if_blank:nTF {#2}
      { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
      {
        \int_compare:nNnTF {#1} = \c_zero
          { \exp_args:No \__clist_item_n_end:n {#2} }
          {
            \exp_args:Nf \__clist_item_n_loop:nw
              { \int_eval:n { #1 - 1 } }
              \prg_do_nothing:
          }
      }
  }
\cs_new:Npn \__clist_item_n_end:n #1 #2 \q_stop
  {
    \__tl_trim_spaces:nn { \q_mark #1 }
      { \exp_last_unbraced:No \__clist_item_n_strip:w } ,
  }
```

```
\cs_new:Npn \__clist_item_n_strip:w #1 , { \exp_not:n {#1} }
```

(*End definition for* `\clist_item:nn`*. This function is documented on page* **??**.)

`\clist_set_from_seq:NN`
`\clist_set_from_seq:cN`
`\clist_set_from_seq:Nc`
`\clist_set_from_seq:cc`
`\clist_gset_from_seq:NN`
`\clist_gset_from_seq:cN`
`\clist_gset_from_seq:Nc`
`\clist_gset_from_seq:cc`
`\__clist_set_from_seq:NNNN`
`\__clist_wrap_item:n`
`\__clist_set_from_seq:w`

Setting a comma list from a comma-separated list is done using a simple mapping. We wrap most items with `\exp_not:n`, and a comma. Items which contain a comma or a space are surrounded by an extra set of braces. The first comma must be removed, except in the case of an empty comma-list.

```
\cs_new_protected:Npn \clist_set_from_seq:NN
  { \__clist_set_from_seq:NNNN \clist_clear:N  \tl_set:Nx  }
\cs_new_protected:Npn \clist_gset_from_seq:NN
  { \__clist_set_from_seq:NNNN \clist_gclear:N \tl_gset:Nx }
\cs_new_protected:Npn \__clist_set_from_seq:NNNN #1#2#3#4
  {
    \seq_if_empty:NTF #4
      { #1 #3 }
      {
        #2 #3
          {
            \exp_last_unbraced:Nf \use_none:n
              { \seq_map_function:NN #4 \__clist_wrap_item:n }
          }
      }
  }
\cs_new:Npn \__clist_wrap_item:n #1
  {
    ,
    \tl_if_empty:oTF { \__clist_set_from_seq:w #1 ~ , #1 ~ }
      { \exp_not:n   {#1}   }
      { \exp_not:n { {#1} } }
  }
\cs_new:Npn \__clist_set_from_seq:w #1 , #2 ~ { }
\cs_generate_variant:Nn \clist_set_from_seq:NN  {     Nc }
\cs_generate_variant:Nn \clist_set_from_seq:NN  { c , cc }
\cs_generate_variant:Nn \clist_gset_from_seq:NN {     Nc }
\cs_generate_variant:Nn \clist_gset_from_seq:NN { c , cc }
```

(*End definition for* `\clist_set_from_seq:NN` *and others. These functions are documented on page* **??**.)

`\clist_const:Nn`
`\clist_const:cn`
`\clist_const:Nx`
`\clist_const:cx`

Creating and initializing a constant comma list is done in a way similar to `\clist_set:Nn` and `\clist_gset:Nn`, being careful to strip spaces.

```
\cs_new_protected:Npn \clist_const:Nn #1#2
  { \tl_const:Nx #1 { \__clist_trim_spaces:n {#2} } }
\cs_generate_variant:Nn \clist_const:Nn { c , Nx , cx }
```

(*End definition for* `\clist_const:Nn` *and others. These functions are documented on page* **??**.)

`\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.

```
\prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
  {
    \__clist_if_empty_n:w ? #1
    , \q_mark \prg_return_false:
    , \q_mark \prg_return_true:
    \q_stop
  }
\cs_new:Npn \__clist_if_empty_n:w #1 ,
  {
    \tl_if_empty:oTF { \use_none:nn #1 ? }
      { \__clist_if_empty_n:w ? }
      { \__clist_if_empty_n:wNw }
  }
\cs_new:Npn \__clist_if_empty_n:wNw #1 \q_mark #2#3 \q_stop {#2}
```

(*End definition for* \clist_if_empty:n*. These functions are documented on page 195.*)

36.5 Additions to l3coffins

```
⟨@@=coffin⟩
```

36.6 Rotating coffins

\l__coffin_sin_fp
\l__coffin_cos_fp

Used for rotations to get the sine and cosine values.

```
\fp_new:N \l__coffin_sin_fp
\fp_new:N \l__coffin_cos_fp
```

(*End definition for* \l__coffin_sin_fp*. This variable is documented on page* ??.)

\l__coffin_bounding_prop

A property list for the bounding box of a coffin. This is only needed during the rotation, so there is just the one.

```
\prop_new:N \l__coffin_bounding_prop
```

(*End definition for* \l__coffin_bounding_prop*. This variable is documented on page* ??.)

\l__coffin_bounding_shift_dim

The shift of the bounding box of a coffin from the real content.

```
\dim_new:N \l__coffin_bounding_shift_dim
```

(*End definition for* \l__coffin_bounding_shift_dim*. This variable is documented on page* ??.)

\l__coffin_left_corner_dim
\l__coffin_right_corner_dim
\l__coffin_bottom_corner_dim
\l__coffin_top_corner_dim

These are used to hold maxima for the various corner values: these thus define the minimum size of the bounding box after rotation.

```
\dim_new:N \l__coffin_left_corner_dim
\dim_new:N \l__coffin_right_corner_dim
\dim_new:N \l__coffin_bottom_corner_dim
\dim_new:N \l__coffin_top_corner_dim
```

(*End definition for* \l__coffin_left_corner_dim*. This variable is documented on page* ??.)

\coffin_rotate:Nn
\coffin_rotate:cn

Rotating a coffin requires several steps which can be conveniently run together. The sine and cosine of the angle in degrees are computed. This is then used to set \l__coffin_sin_fp and \l__coffin_cos_fp, which are carried through unchanged for the rest of the procedure.

```
\cs_new_protected:Npn \coffin_rotate:Nn #1#2
  {
    \fp_set:Nn \l__coffin_sin_fp { sind ( #2 ) }
    \fp_set:Nn \l__coffin_cos_fp { cosd ( #2 ) }
```

The corners and poles of the coffin can now be rotated around the origin. This is best achieved using mapping functions.

```
    \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
      { \__coffin_rotate_corner:Nnnn #1 {##1} ##2 }
    \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
      { \__coffin_rotate_pole:Nnnnnn #1 {##1} ##2 }
```

The bounding box of the coffin needs to be rotated, and to do this the corners have to be found first. They are then rotated in the same way as the corners of the coffin material itself.

```
    \__coffin_set_bounding:N #1
    \prop_map_inline:Nn \l__coffin_bounding_prop
      { \__coffin_rotate_bounding:nnn {##1} ##2 }
```

At this stage, there needs to be a calculation to find where the corners of the content and the box itself will end up.

```
    \__coffin_find_corner_maxima:N #1
    \__coffin_find_bounding_shift:
    \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.

```
    \hbox_set:Nn \l__coffin_internal_box
      {
        \tex_kern:D
          \__dim_eval:w
            \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim
          \__dim_eval_end:
        \box_move_down:nn { \l__coffin_bottom_corner_dim }
          { \box_use:N #1 }
      }
```

If there have been any previous rotations then the size of the bounding box will be bigger than the contents. This can be corrected easily by setting the size of the box to the height and width of the content. As this operation requires setting box dimensions and these transcend grouping, the safe way to do this is to use the internal box and to reset the result into the target box.

```
    \box_set_ht:Nn \l__coffin_internal_box
      { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
    \box_set_dp:Nn \l__coffin_internal_box { 0 pt }
    \box_set_wd:Nn \l__coffin_internal_box
      { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
    \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.

```
    \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
      { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
    \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
      { \__coffin_shift_pole:Nnnnnn #1 {##1} ##2 }
  }
\cs_generate_variant:Nn \coffin_rotate:Nn { c }
```

(*End definition for* `\coffin_rotate:Nn` *and* `\coffin_rotate:cn`*. These functions are documented on page* **??**.)

`\__coffin_set_bounding:N`

The bounding box corners for a coffin are easy enough to find: this is the same code as for the corners of the material itself, but using a dedicated property list.

```
\cs_new_protected:Npn \__coffin_set_bounding:N #1
  {
    \prop_put:Nnx \l__coffin_bounding_prop { tl }
      { { 0 pt } { \dim_use:N \box_ht:N #1 } }
    \prop_put:Nnx \l__coffin_bounding_prop { tr }
      { { \dim_use:N \box_wd:N #1 } { \dim_use:N \box_ht:N #1 } }
    \dim_set:Nn \l__coffin_internal_dim { - \box_dp:N #1 }
    \prop_put:Nnx \l__coffin_bounding_prop { bl }
      { { 0 pt } { \dim_use:N \l__coffin_internal_dim } }
    \prop_put:Nnx \l__coffin_bounding_prop { br }
      { { \dim_use:N \box_wd:N #1 } { \dim_use:N \l__coffin_internal_dim } }
  }
```

(*End definition for* `\__coffin_set_bounding:N`*. This function is documented on page* **??**.)

`\__coffin_rotate_bounding:nnn`
`\__coffin_rotate_corner:Nnnn`

Rotating the position of the corner of the coffin is just a case of treating this as a vector from the reference point. The same treatment is used for the corners of the material itself and the bounding box.

```
\cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
  {
    \__coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
    \prop_put:Nnx \l__coffin_bounding_prop {#1}
      { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
  }
\cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
  {
    \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
      { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
  }
```

(*End definition for* `\__coffin_rotate_bounding:nnn`. *This function is documented on page* **??**.)

`\__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.

```
\cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
  {
    \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
    \__coffin_rotate_vector:nnNN {#5} {#6}
      \l__coffin_x_prime_dim \l__coffin_y_prime_dim
    \__coffin_set_pole:Nnx #1 {#2}
      {
        { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
        { \dim_use:N \l__coffin_x_prime_dim }
        { \dim_use:N \l__coffin_y_prime_dim }
      }
  }
```

(*End definition for* `\__coffin_rotate_pole:Nnnnnn`. *This function is documented on page* **??**.)

`\__coffin_rotate_vector:nnNN`

A rotation function, which needs only an input vector (as dimensions) and an output space. The values `\l__coffin_cos_fp` and `\l__coffin_sin_fp` should previously have been set up correctly. Working this way means that the floating point work is kept to a minimum: for any given rotation the sin and cosine values do no change, after all.

```
\cs_new_protected:Npn \__coffin_rotate_vector:nnNN #1#2#3#4
  {
    \dim_set:Nn #3
      {
        \fp_to_dim:n
          {
                \dim_to_fp:n {#1} * \l__coffin_cos_fp
            - ( \dim_to_fp:n {#2} * \l__coffin_sin_fp )
          }
      }
    \dim_set:Nn #4
      {
        \fp_to_dim:n
          {
                \dim_to_fp:n {#1} * \l__coffin_sin_fp
            + ( \dim_to_fp:n {#2} * \l__coffin_cos_fp )
          }
      }
  }
```

(*End definition for* `\__coffin_rotate_vector:nnNN`. *This function is documented on page* **??**.)

`\__coffin_find_corner_maxima:N`
`\__coffin_find_corner_maxima_aux:nn`

The idea here is to find the extremities of the content of the coffin. This is done by looking for the smallest values for the bottom and left corners, and the largest values for the top and right corners. The values start at the maximum dimensions so that the case where all are positive or all are negative works out correctly.

```
\cs_new_protected:Npn \__coffin_find_corner_maxima:N #1
```

```
  {
    \dim_set:Nn \l__coffin_top_corner_dim   { -\c_max_dim }
    \dim_set:Nn \l__coffin_right_corner_dim { -\c_max_dim }
    \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
    \dim_set:Nn \l__coffin_left_corner_dim   { \c_max_dim }
    \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
      { \__coffin_find_corner_maxima_aux:nn ##2 }
  }
\cs_new_protected:Npn \__coffin_find_corner_maxima_aux:nn #1#2
  {
    \dim_set:Nn \l__coffin_left_corner_dim
     { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
    \dim_set:Nn \l__coffin_right_corner_dim
     { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
    \dim_set:Nn \l__coffin_bottom_corner_dim
     { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
    \dim_set:Nn \l__coffin_top_corner_dim
     { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
  }
```

(*End definition for* `\__coffin_find_corner_maxima:N`*. This function is documented on page* **??**.)

`\__coffin_find_bounding_shift:`
`\__coffin_find_bounding_shift_aux:nn`

The approach to finding the shift for the bounding box is similar to that for the corners. However, there is only one value needed here and a fixed input property list, so things are a bit clearer.

```
\cs_new_protected_nopar:Npn \__coffin_find_bounding_shift:
  {
    \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }
    \prop_map_inline:Nn \l__coffin_bounding_prop
      { \__coffin_find_bounding_shift_aux:nn ##2 }
  }
\cs_new_protected:Npn \__coffin_find_bounding_shift_aux:nn #1#2
  {
    \dim_set:Nn \l__coffin_bounding_shift_dim
      { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
  }
```

(*End definition for* `\__coffin_find_bounding_shift:`*. This function is documented on page* **??**.)

`\__coffin_shift_corner:Nnnn`
`\__coffin_shift_pole:Nnnnnn`

Shifting the corners and poles of a coffin means subtracting the appropriate values from the x- and y-components. For the poles, this means that the direction vector is unchanged.

```
\cs_new_protected:Npn \__coffin_shift_corner:Nnnn #1#2#3#4
  {
    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _ prop } {#2}
      {
        { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
        { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
      }
  }
\cs_new_protected:Npn \__coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
```

```
  {
    \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _ prop } {#2}
      {
        { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
        { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
        {#5} {#6}
      }
  }
```

(*End definition for* `\__coffin_shift_corner:Nnnn`. *This function is documented on page* ??.)

36.7 Resizing coffins

`\l__coffin_scale_x_fp`
`\l__coffin_scale_y_fp`

Storage for the scaling factors in x and y, respectively.

```
\fp_new:N \l__coffin_scale_x_fp
\fp_new:N \l__coffin_scale_y_fp
```

(*End definition for* `\l__coffin_scale_x_fp`. *This variable is documented on page* ??.)

`\l__coffin_scaled_total_height_dim`
`\l__coffin_scaled_width_dim`

When scaling, the values given have to be turned into absolute values.

```
\dim_new:N \l__coffin_scaled_total_height_dim
\dim_new:N \l__coffin_scaled_width_dim
```

(*End definition for* `\l__coffin_scaled_total_height_dim`. *This variable is documented on page* ??.)

`\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.

```
\cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
  {
    \fp_set:Nn \l__coffin_scale_x_fp
      { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
    \fp_set:Nn \l__coffin_scale_y_fp
      {
        \dim_to_fp:n {#3} / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
      }
    \box_resize:Nnn #1 {#2} {#3}
    \__coffin_resize_common:Nnn #1 {#2} {#3}
  }
\cs_generate_variant:Nn \coffin_resize:Nnn { c }
```

(*End definition for* `\coffin_resize:Nnn` *and* `\coffin_resize:cnn`. *These functions are documented on page* ??.)

`\__coffin_resize_common:Nnn`

The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```
\cs_new_protected:Npn \__coffin_resize_common:Nnn #1#2#3
  {
    \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
      { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }
    \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
      { \__coffin_scale_pole:Nnnnnn #1 {##1} ##2 }
```

Negative x-scaling values will place the poles in the wrong location: this is corrected here.

```
    \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
      {
        \prop_map_inline:cn { l__coffin_corners_ \__int_value:w #1 _prop }
          { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
        \prop_map_inline:cn { l__coffin_poles_ \__int_value:w #1 _prop }
          { \__coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
      }
  }
```

(*End definition for* `\__coffin_resize_common:Nnn`*. This function is documented on page* **??**.)

`\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.

```
\cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
  {
    \fp_set:Nn \l__coffin_scale_x_fp {#2}
    \fp_set:Nn \l__coffin_scale_y_fp {#3}
    \box_scale:Nnn #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }
    \dim_set:Nn \l__coffin_internal_dim
      { \coffin_ht:N #1 + \coffin_dp:N #1 }
    \dim_set:Nn \l__coffin_scaled_total_height_dim
      { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_internal_dim }
    \dim_set:Nn \l__coffin_scaled_width_dim
      { -\fp_abs:n { \l__coffin_scale_x_fp  } \coffin_wd:N #1 }
    \__coffin_resize_common:Nnn #1
      { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
  }
\cs_generate_variant:Nn \coffin_scale:Nnn { c }
```

(*End definition for* `\coffin_scale:Nnn` *and* `\coffin_scale:cnn`*. These functions are documented on page* **??**.)

`\__coffin_scale_vector:nnNN`

This functions scales a vector from the origin using the pre-set scale factors in x and y. This is a much less complex operation than rotation, and as a result the code is a lot clearer.

```
\cs_new_protected:Npn \__coffin_scale_vector:nnNN #1#2#3#4
  {
    \dim_set:Nn #3
      { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
    \dim_set:Nn #4
      { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
  }
```

(*End definition for* `\__coffin_scale_vector:nnNN`*. This function is documented on page* **??**.)

`\__coffin_scale_corner:Nnnn`
`\__coffin_scale_pole:Nnnnnn`

Scaling both corners and poles is a simple calculation using the preceding vector scaling.

```
\cs_new_protected:Npn \__coffin_scale_corner:Nnnn #1#2#3#4
```

```
  {
    \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
      { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
  }
\cs_new_protected:Npn \__coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
  {
    \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
    \__coffin_set_pole:Nnx #1 {#2}
      {
        { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
        {#5} {#6}
      }
  }
```

(*End definition for* `\__coffin_scale_corner:Nnnn`*. This function is documented on page* ??.)

`\__coffin_x_shift_corner:Nnnn`
`\__coffin_x_shift_pole:Nnnnnn`

These functions correct for the x displacement that takes place with a negative horizontal scaling.

```
\cs_new_protected:Npn \__coffin_x_shift_corner:Nnnn #1#2#3#4
  {
    \prop_put:cnx { l__coffin_corners_ \__int_value:w #1 _prop } {#2}
      {
        { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
      }
  }
\cs_new_protected:Npn \__coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
  {
    \prop_put:cnx { l__coffin_poles_ \__int_value:w #1 _prop } {#2}
      {
        { \dim_eval:n #3 + \box_wd:N #1 } {#4}
        {#5} {#6}
      }
  }
```

(*End definition for* `\__coffin_x_shift_corner:Nnnn`*. This function is documented on page* ??.)

36.8 Additions to l3file

```
⟨@@=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.

```
\cs_new_nopar:Npn \ior_map_break:
  { \__prg_map_break:Nn \ior_map_break: { } }
\cs_new_nopar:Npn \ior_map_break:n
  { \__prg_map_break:Nn \ior_map_break: }
```

(*End definition for* `\ior_map_break:` *and* `\ior_map_break:n`*. These functions are documented on page 197.*)

\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".

```
\cs_new_protected_nopar:Npn \ior_map_inline:Nn
  { \__ior_map_inline:NNn \ior_get:NN }
\cs_new_protected_nopar:Npn \ior_str_map_inline:Nn
  { \__ior_map_inline:NNn \ior_get_str:NN }
\cs_new_protected_nopar:Npn \__ior_map_inline:NNn
  {
    \int_gincr:N \g__prg_map_int
    \exp_args:Nc \__ior_map_inline:NNNn
      { __prg_map_ \int_use:N \g__prg_map_int :n }
  }
\cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4
  {
    \cs_set:Npn #1 ##1 {#4}
    \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
    \__prg_break_point:Nn \ior_map_break:
      { \int_gdecr:N \g__prg_map_int }
  }
\cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
  {
    #2 #3 \l__ior_internal_tl
    \ior_if_eof:NF #3
      {
        \exp_args:No #1 \l__ior_internal_tl
        \__ior_map_inline_loop:NNN #1#2#3
      }
  }
\tl_new:N  \l__ior_internal_tl
```

(*End definition for* `\ior_map_inline:Nn` *and* `\ior_str_map_inline:Nn`*. These functions are documented on page* ??.)

36.9 Additions to l3fp

```
⟨@@=fp⟩
```

\fp_set_from_dim:Nn
\fp_set_from_dim:cn
\fp_gset_from_dim:Nn
\fp_gset_from_dim:cn

Use the appropriate function from l3fp-convert.

```
\cs_new_protected:Npn \fp_set_from_dim:Nn #1#2
  { \tl_set:Nx #1 { \dim_to_fp:n {#2} } }
\cs_new_protected:Npn \fp_gset_from_dim:Nn #1#2
  { \tl_gset:Nx #1 { \dim_to_fp:n {#2} } }
\cs_generate_variant:Nn \fp_set_from_dim:Nn  { c }
\cs_generate_variant:Nn \fp_gset_from_dim:Nn { c }
```

(*End definition for* `\fp_set_from_dim:Nn` *and others. These functions are documented on page* ??.)

36.10 Additions to l3prop

```
⟨@@=prop⟩
```

\prop_map_tokens:Nn
\prop_map_tokens:cn
__prop_map_tokens:nwwn

The mapping is very similar to `\prop_map_function:NN`. It grabs one key–value pair at a time, and stops when reaching the marker key `\q_recursion_tail`, which cannot 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.

```
\cs_new:Npn \prop_map_tokens:Nn #1#2
  {
    \exp_last_unbraced:Nno \__prop_map_tokens:nwwn {#2} #1
      \__prop_pair:wn \q_recursion_tail \s__prop { }
    \__prg_break_point:Nn \prop_map_break: { }
  }
\cs_new:Npn \__prop_map_tokens:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
  {
    \if_meaning:w \q_recursion_tail #3
      \exp_after:wN \prop_map_break:
    \fi:
    \use:n {#1} {#3} {#4}
    \__prop_map_tokens:nwwn {#1}
  }
\cs_generate_variant:Nn \prop_map_tokens:Nn { c }
```

(*End definition for* `\prop_map_tokens:Nn` *and* `\prop_map_tokens:cn`*. These functions are documented on page* **??**.)

\prop_get:Nn
\prop_get:cn
__prop_get_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_get_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.

```
\cs_new:Npn \prop_get:Nn #1#2
  {
    \exp_last_unbraced:Noo \__prop_get_Nn:nwwn { \tl_to_str:n {#2} } #1
      \__prop_pair:wn \tl_to_str:n {#2} \s__prop { }
    \__prg_break_point:
  }
\cs_new:Npn \__prop_get_Nn:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
  {
    \str_if_eq_x:nnTF {#1} {#3}
      { \__prg_break:n { \exp_not:n {#4} } }
      { \__prop_get_Nn:nwwn {#1} }
  }
\cs_generate_variant:Nn \prop_get:Nn { c }
```

(*End definition for* `\prop_get:Nn` *and* `\prop_get:cn`*. These functions are documented on page* **??**.)

36.11 Additions to l3seq

```
⟨@@=seq⟩
```

\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.

```
\cs_new:Npn \seq_item:Nn #1
  { \exp_after:wN \__seq_item:wNn #1 \q_stop #1 }
\cs_new:Npn \__seq_item:wNn \s__seq #1 \q_stop #2#3
  {
    \exp_args:Nf \__seq_item:nnn
      {
        \int_eval:n
          {
            \int_compare:nNnT {#3} < \c_zero
              { \seq_count:N #2 + \c_one + }
            #3
          }
      }
    #1
    { ? \__prg_break: } { }
    \__prg_break_point:
  }
\cs_new:Npn \__seq_item:nnn #1#2#3
  {
    \use_none:n #2
    \int_compare:nNnTF {#1} = \c_one
      { \__prg_break:n { \exp_not:n {#3} } }
      { \exp_args:Nf \__seq_item:nnn { \int_eval:n { #1 - 1 } } }
  }
\cs_generate_variant:Nn \seq_item:Nn { c }
```

(*End definition for* `\seq_item:Nn` *and* `\seq_item:cn`*. These functions are documented on page* **??**.)

\seq_mapthread_function:NNN
\seq_mapthread_function:NcN
\seq_mapthread_function:cNN
\seq_mapthread_function:ccN
__seq_mapthread_function:wNN
__seq_mapthread_function:wNw
__seq_mapthread_function:Nnnwnn

The idea here is to first expand both sequences, adding the usual `{ ? \__prg_break: } { }` to the end of each one. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first tokens of `#2` and `#5`, which for items in the sequences will both be `\s__seq \__seq_item:n`. The function to be mapped will then be applied to the two entries. When the code hits the end of one of the sequences, the break material will stop the entire loop and tidy up. This avoids needing to find the count of the two sequences, or worrying about which is longer.

```
\cs_new:Npn \seq_mapthread_function:NNN #1#2#3
  { \exp_after:wN \__seq_mapthread_function:wNN #2 \q_stop #1 #3 }
\cs_new:Npn \__seq_mapthread_function:wNN \s__seq #1 \q_stop #2#3
  {
    \exp_after:wN \__seq_mapthread_function:wNw #2 \q_stop #3
      #1 { ? \__prg_break: } { }
    \__prg_break_point:
```

```
  }
\cs_new:Npn \__seq_mapthread_function:wNw \s__seq #1 \q_stop #2
  {
    \__seq_mapthread_function:Nnnwnn #2
      #1 { ? \__prg_break: } { }
    \q_stop
  }
\cs_new:Npn \__seq_mapthread_function:Nnnwnn #1#2#3#4 \q_stop #5#6
  {
    \use_none:n #2
    \use_none:n #5
    #1 {#3} {#6}
    \__seq_mapthread_function:Nnnwnn #1 #4 \q_stop
  }
\cs_generate_variant:Nn \seq_mapthread_function:NNN {     Nc }
\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* **??**.)

\seq_set_from_clist:NN
\seq_set_from_clist:cN
\seq_set_from_clist:Nc
\seq_set_from_clist:cc
\seq_set_from_clist:Nn
\seq_set_from_clist:cn
\seq_gset_from_clist:NN
\seq_gset_from_clist:cN
\seq_gset_from_clist:Nc
\seq_gset_from_clist:cc
\seq_gset_from_clist:Nn
\seq_gset_from_clist:cn

Setting a sequence from a comma-separated list is done using a simple mapping.

```
\cs_new_protected:Npn \seq_set_from_clist:NN #1#2
  {
    \tl_set:Nx #1
      { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
  }
\cs_new_protected:Npn \seq_set_from_clist:Nn #1#2
  {
    \tl_set:Nx #1
      { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
  }
\cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
  {
    \tl_gset:Nx #1
      { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
  }
\cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
  {
    \tl_gset:Nx #1
      { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
  }
\cs_generate_variant:Nn \seq_set_from_clist:NN  {     Nc }
\cs_generate_variant:Nn \seq_set_from_clist:NN  { c , cc }
\cs_generate_variant:Nn \seq_set_from_clist:Nn  { c      }
\cs_generate_variant:Nn \seq_gset_from_clist:NN {     Nc }
\cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
\cs_generate_variant:Nn \seq_gset_from_clist:Nn { c      }
```

(*End definition for* `\seq_set_from_clist:NN` *and others. These functions are documented on page* **??**.)

\seq_reverse:N
\seq_reverse:c
\seq_greverse:N
\seq_greverse:c
__seq_reverse:NN
__seq_reverse_item:nwn

Previously, `\seq_reverse:N` was coded by collecting the items in reverse order after an `\exp_stop_f:` marker.

```
\cs_new_protected:Npn \seq_reverse:N #1
  {
    \cs_set_eq:NN \@@_item:n \@@_reverse_item:nw
    \tl_set:Nf #2 { #2 \exp_stop_f: }
  }
\cs_new:Npn \@@_reverse_item:nw #1 #2 \exp_stop_f:
  {
    #2 \exp_stop_f:
    \@@_item:n {#1}
  }
```

At first, this seems optimal, since we can forget about each item as soon as it is placed after `\exp_stop_f:`. Unfortunately, TeX's usual tail recursion does not take place in this case: since the following `\__seq_reverse_item:nw` only reads tokens until `\exp_stop_f:`, and never reads the `\@@_item:n {#1}` left by the previous call, TeX cannot remove that previous call from the stack, and in particular must retain the various macro parameters in memory, until the end of the replacement text is reached. The stack is thus only flushed after all the `\__seq_reverse_item:nw` are expanded. Keeping track of the arguments of all those calls uses up a memory quadratic in the length of the sequence. TeX can then not cope with more than a few thousand items.

Instead, we collect the items in the argument of `\exp_not:n`. The previous calls are cleanly removed from the stack, and the memory consumption becomes linear.

```
\cs_new_protected_nopar:Npn \seq_reverse:N
  { \__seq_reverse:NN \tl_set:Nx }
\cs_new_protected_nopar:Npn \seq_greverse:N
  { \__seq_reverse:NN \tl_gset:Nx }
\cs_new_protected:Npn \__seq_reverse:NN #1 #2
  {
    \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
    \cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn
    #1 #2 { #2 \exp_not:n { } }
    \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
  }
\cs_new:Npn \__seq_reverse_item:nwn #1 #2 \exp_not:n #3
  {
    #2
    \exp_not:n { \__seq_item:n {#1} #3 }
  }
\cs_generate_variant:Nn \seq_reverse:N  { c }
\cs_generate_variant:Nn \seq_greverse:N { c }
```

(*End definition for* `\seq_reverse:N` *and others. These functions are documented on page* ??.)

`\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.

```
\cs_new_protected_nopar:Npn \seq_set_filter:NNn
```

```
  { \__seq_set_filter:NNNn \tl_set:Nx }
\cs_new_protected_nopar:Npn \seq_gset_filter:NNn
  { \__seq_set_filter:NNNn \tl_gset:Nx }
\cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
  {
    \__seq_push_item_def:n { \bool_if:nT {#4} { \__seq_wrap_item:n {##1} } }
    #1 #2 { #3 }
    \__seq_pop_item_def:
  }
```

(*End definition for* `\seq_set_filter:NNn` *and* `\seq_gset_filter:NNn`*. These functions are documented on page* 200*.*)

`\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.

```
\cs_new_protected_nopar:Npn \seq_set_map:NNn
  { \__seq_set_map:NNNn \tl_set:Nx }
\cs_new_protected_nopar:Npn \seq_gset_map:NNn
  { \__seq_set_map:NNNn \tl_gset:Nx }
\cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
  {
    \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
    #1 #2 { #3 }
    \__seq_pop_item_def:
  }
```

(*End definition for* `\seq_set_map:NNn` *and* `\seq_gset_map:NNn`*. These functions are documented on page* 201*.*)

36.12 Additions to l3skip

```
⟨@@=dim⟩
```

`\dim_to_pt:n`

A copy of the internal function `\__dim_strip_pt:n`, which should perhaps be eliminated in favor of `\dim_to_pt:n`.

```
\cs_new_eq:NN \dim_to_pt:n \__dim_strip_pt:n
```

(*End definition for* `\dim_to_pt:n`*. This function is documented on page* 201*.*)

`\dim_to_unit:nn`
`\__dim_to_unit:n`

An analog of `\dim_ratio:nn` that produces a decimal number as its result, rather than a rational fraction for use within dimension expressions. The naive implementation as

```
\cs_new:Npn \dim_to_unit:nn #1#2
  { \dim_to_pt:n { 1pt * \dim_ratio:nn {#1} {#2} } }
```

would not ignore trailing tokens (see documentation), so we need a bit more work.

```
\cs_new:Npn \dim_to_unit:nn #1#2
  {
    \dim_to_pt:n
      {
        1pt * \__dim_to_unit:n { \dim_to_pt:n {#1} pt }
        / \__dim_to_unit:n { \dim_to_pt:n {#2} pt }
```

```
      }
  }
\cs_new:Npn \__dim_to_unit:n #1
  { \__int_value:w \__dim_eval:w #1 \__dim_eval_end: }
```

(*End definition for* `\dim_to_unit:nn`*. This function is documented on page 201.*)

```
⟨@@=skip⟩
```

\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.

```
\cs_new:Npn \skip_split_finite_else_action:nnNN #1#2#3#4
  {
    \skip_if_finite:nTF {#1}
      {
        #3 = \etex_gluestretch:D #1 \scan_stop:
        #4 = \etex_glueshrink:D  #1 \scan_stop:
      }
      {
        #3 = \c_zero_skip
        #4 = \c_zero_skip
        #2
      }
  }
```

(*End definition for* `\skip_split_finite_else_action:nnNN`*. This function is documented on page 201.*)

36.13 Additions to l3tl

```
⟨@@=tl⟩
```

\tl_if_single_token_p:n
\tl_if_single_token:nTF

There are four cases: empty token list, token list starting with a normal token, with a brace group, or with a space token. If the token list starts with a normal token, remove it and check for emptiness. Otherwise, compare with a single space, only case where we have a single token.

```
\prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
  {
    \tl_if_head_is_N_type:nTF {#1}
      { \__str_if_eq_x_return:nn { \exp_not:o { \use_none:n #1 } } { } }
      { \__str_if_eq_x_return:nn { \exp_not:n {#1} } { ~ } }
  }
```

(*End definition for* `\tl_if_single_token:n`*. These functions are documented on page 202.*)

\tl_reverse_tokens:n
__tl_reverse_group:nn

The same as `\tl_reverse:n` but with recursion within brace groups.

```
\cs_new:Npn \tl_reverse_tokens:n #1
  {
    \etex_unexpanded:D \exp_after:wN
      {
        \tex_romannumeral:D
```

```
        \__tl_act:NNNnn
          \__tl_reverse_normal:nN
          \__tl_reverse_group:nn
          \__tl_reverse_space:n
          { }
          {#1}
      }
  }
\cs_new:Npn \__tl_reverse_group:nn #1
  {
    \__tl_act_group_recurse:Nnn
      \__tl_act_reverse_output:n
      { \tl_reverse_tokens:n }
  }
```

__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.

```
\cs_new:Npn \__tl_act_group_recurse:Nnn #1#2#3
  {
    \exp_args:Nf #1
      { \exp_after:wN \exp_after:wN \exp_after:wN { #2 {#3} } }
  }
```

(*End definition for* `\tl_reverse_tokens:n`*. This function is documented on page 202.*)

\tl_count_tokens:n
__tl_act_count_normal:nN
__tl_act_count_group:nn
__tl_act_count_space:n

The token count is computed through an `\int_eval:n` construction. Each `1+` is output to the *left*, into the integer expression, and the sum is ended by the `\c_zero` inserted by `\__tl_act_end:wn`. Somewhat a hack.

```
\cs_new:Npn \tl_count_tokens:n #1
  {
    \int_eval:n
      {
        \__tl_act:NNNnn
          \__tl_act_count_normal:nN
          \__tl_act_count_group:nn
          \__tl_act_count_space:n
          { }
          {#1}
      }
  }
\cs_new:Npn \__tl_act_count_normal:nN #1 #2 { 1 + }
\cs_new:Npn \__tl_act_count_space:n #1 { 1 + }
\cs_new:Npn \__tl_act_count_group:nn #1 #2
  { 2 + \tl_count_tokens:n {#2} + }
```

(*End definition for* `\tl_count_tokens:n`*. This function is documented on page 202.*)

\c__tl_act_uppercase_tl
\c__tl_act_lowercase_tl

These constants contain the correspondence between lowercase and uppercase letters, in the form `aAbBcC...` and `AaBbCc...` respectively.

```
\tl_const:Nn \c__tl_act_uppercase_tl
```

```
  {
    aA bB cC dD eE fF gG hH iI jJ kK lL mM
    nN oO pP qQ rR sS tT uU vV wW xX yY zZ
  }
\tl_const:Nn \c__tl_act_lowercase_tl
  {
    Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm
    Nn Oo Pp Qq Rr Ss Tt Uu Vv Ww Xx Yy Zz
  }
```

(*End definition for* `\c__tl_act_uppercase_tl` *and* `\c__tl_act_lowercase_tl`*. These variables are documented on page* **??**.)

`\tl_expandable_uppercase:n`
`\tl_expandable_lowercase:n`
`\__tl_act_case_normal:nN`
`\__tl_act_case_group:nn`
`\__tl_act_case_space:n`

The only difference between uppercasing and lowercasing is the table of correspondence that is used. As for other token list actions, we feed `\__tl_act:NNNnn` three functions, and this time, we use the ⟨*parameters*⟩ argument to carry which case-changing we are applying. A space is simply output. A normal token is compared to each letter in the alphabet using `\str_if_eq:nn` tests, and converted if necessary to upper/lowercase, before being output. For a group, we must perform the conversion within the group (the `\exp_after:wN` trigger `\romannumeral`, which expands fully to give the converted group), then output.

```
\cs_new:Npn \tl_expandable_uppercase:n #1
  {
    \etex_unexpanded:D \exp_after:wN
      {
        \tex_romannumeral:D
          \__tl_act_case_aux:nn { \c__tl_act_uppercase_tl } {#1}
      }
  }
\cs_new:Npn \tl_expandable_lowercase:n #1
  {
    \etex_unexpanded:D \exp_after:wN
      {
        \tex_romannumeral:D
          \__tl_act_case_aux:nn { \c__tl_act_lowercase_tl } {#1}
      }
  }
\cs_new:Npn \__tl_act_case_aux:nn
  {
    \__tl_act:NNNnn
      \__tl_act_case_normal:nN
      \__tl_act_case_group:nn
      \__tl_act_case_space:n
  }
\cs_new:Npn \__tl_act_case_space:n #1 { \__tl_act_output:n {~} }
\cs_new:Npn \__tl_act_case_normal:nN #1 #2
  {
    \exp_args:Nf \__tl_act_output:n
      {
        \exp_args:NNo \str_case:nnF #2 {#1}
```

```
        { \exp_stop_f: #2 }
      }
  }
\cs_new:Npn \__tl_act_case_group:nn #1 #2
  {
    \exp_after:wN \__tl_act_output:n \exp_after:wN
      { \exp_after:wN { \tex_romannumeral:D \__tl_act_case_aux:nn {#1} {#2} } }
  }
```

(*End definition for* `\tl_expandable_uppercase:n` *and* `\tl_expandable_lowercase:n`*. These functions are documented on page 202.*)

`\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.

```
\cs_new:Npn \tl_item:nn #1#2
  {
    \exp_args:Nf \__tl_item:nn
      {
        \int_eval:n
          {
            \int_compare:nNnT {#2} < \c_zero
              { \tl_count:n {#1} + \c_one + }
            #2
          }
      }
    #1
    \q_recursion_tail
    \__prg_break_point:
  }
\cs_new:Npn \__tl_item:nn #1#2
  {
    \__quark_if_recursion_tail_break:nN {#2} \__prg_break:
    \int_compare:nNnTF {#1} = \c_one
      { \__prg_break:n { \exp_not:n {#2} } }
      { \exp_args:Nf \__tl_item:nn { \int_eval:n { #1 - 1 } } }
  }
\cs_new_nopar:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
\cs_generate_variant:Nn \tl_item:Nn { c }
```

(*End definition for* `\tl_item:nn`*,* `\tl_item:Nn`*, and* `\tl_item:cn`*. These functions are documented on page* **??***.*)

36.14 Additions to l3tokens

```
⟨@@=char⟩
```

`\char_set_active:Npn`
`\char_set_active:Npx`
`\char_gset_active:Npn`
`\char_gset_active:Npx`
`\char_set_active_eq:NN`
`\char_gset_active_eq:NN`

```
\group_begin:
  \char_set_catcode_active:N \^^@
  \cs_set:Npn \char_tmp:NN #1#2
    {
```

```
      \cs_new:Npn #1 ##1
        {
          \char_set_catcode_active:n { `##1 }
          \group_begin:
          \char_set_lccode:nn { `\^^@ } { `##1 }
          \tl_to_lowercase:n { \group_end: #2 ^^@ }
        }
    }
  \char_tmp:NN \char_set_active:Npn    \cs_set:Npn
  \char_tmp:NN \char_set_active:Npx    \cs_set:Npx
  \char_tmp:NN \char_gset_active:Npn   \cs_gset:Npn
  \char_tmp:NN \char_gset_active:Npx   \cs_gset:Npx
  \char_tmp:NN \char_set_active_eq:NN  \cs_set_eq:NN
  \char_tmp:NN \char_gset_active_eq:NN \cs_gset_eq:NN
\group_end:
```

(*End definition for* `\char_set_active:Npn` *and* `\char_set_active:Npx`. *These functions are documented on page 203.*)

```
⟨@@=peek⟩
```

`\peek_N_type:TF`
`\__peek_execute_branches_N_type:`
`\__peek_N_type:w`
`\__peek_N_type_aux:nnw`

All tokens are `N`-type tokens, except in four cases: begin-group tokens, end-group tokens, space tokens with character code 32, and outer tokens. Since `\l_peek_token` might be outer, we cannot use the convenient `\bool_if:nTF` function, and must resort to the old trick of using `\ifodd` to expand a set of tests. The `false` branch of this test is taken if the token is one of the first three kinds of non-`N`-type tokens (explicit or implicit), thus we call `\__peek_false:w`. In the `true` branch, we must detect outer tokens, without impacting performance too much for non-outer tokens. The first filter is to search for `outer` in the `\meaning` of `\l_peek_token`. If that is absent, `\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.

```
\group_begin:
  \char_set_catcode_other:N \O
  \char_set_catcode_other:N \U
  \char_set_catcode_other:N \T
  \char_set_catcode_other:N \E
  \char_set_catcode_other:N \R
  \tl_to_lowercase:n
    {
      \cs_new_protected_nopar:Npn \__peek_execute_branches_N_type:
        {
          \if_int_odd:w
              \if_catcode:w \exp_not:N \l_peek_token {   \c_two \fi:
              \if_catcode:w \exp_not:N \l_peek_token }   \c_two \fi:
              \if_meaning:w \l_peek_token \c_space_token \c_two \fi:
```

```
            \c_one
          \exp_after:wN \__peek_N_type:w
            \token_to_meaning:N \l_peek_token
            \q_mark \__peek_N_type_aux:nnw
            OUTER \q_mark \use_none_delimit_by_q_stop:w
            \q_stop
          \exp_after:wN \__peek_true:w
        \else:
          \exp_after:wN \__peek_false:w
        \fi:
      }
    \cs_new_protected:Npn \__peek_N_type:w #1 OUTER #2 \q_mark #3
      { #3 {#1} {#2} }
  }
\group_end:
\cs_new_protected:Npn \__peek_N_type_aux:nnw #1 #2 #3 \fi:
  {
    \fi:
    \tl_if_in:noTF {#1} { \tl_to_str:n {ma} }
      { \__peek_true:w }
      { \tl_if_empty:nTF {#2} { \__peek_true:w } { \__peek_false:w } }
  }
\cs_new_protected_nopar:Npn \peek_N_type:TF
  { \__peek_token_generic:NNTF \__peek_execute_branches_N_type: \scan_stop: }
\cs_new_protected_nopar:Npn \peek_N_type:T
  { \__peek_token_generic:NNT  \__peek_execute_branches_N_type: \scan_stop: }
\cs_new_protected_nopar:Npn \peek_N_type:F
  { \__peek_token_generic:NNF  \__peek_execute_branches_N_type: \scan_stop: }
```

(*End definition for* `\peek_N_type:TF`*. This function is documented on page 204.*)

```
⟨/initex | package⟩
```

37 l3drivers Implementation

```
⟨*initex | package⟩
⟨@@=driver⟩
```

```
⟨*package⟩
\ProvidesExplFile
⟨*dvipdfmx⟩
  {l3dvidpfmx.def}{\ExplFileDate}{\ExplFileVersion}
  {L3 Experimental driver: dvipdfmx}
⟨/dvipdfmx⟩
⟨*dvips⟩
  {l3dvips.def}{\ExplFileDate}{\ExplFileVersion}
  {L3 Experimental driver: dvips}
⟨/dvips⟩
⟨*pdfmode⟩
  {l3pdfmode.def}{\ExplFileDate}{\ExplFileVersion}
  {L3 Experimental driver: PDF mode}
```

```
⟨/pdfmode⟩
⟨*xdvipdfmx⟩
  {l3xdvidpfmx.def}{\ExplFileDate}{\ExplFileVersion}
  {L3 Experimental driver: xdvipdfmx}
⟨/xdvipdfmx⟩
⟨/package⟩
```

37.1 Settings for direct PDF output

If the driver loaded is `pdfmode` then direct PDF output is required. (This may of course alter: it might be that the driver is picked based on the value of `\pdftex_pdfoutput:D`.)

```
⟨*initex⟩
⟨*pdfmode⟩
\pdftex_pdfoutput:D = 1 \scan_stop:
⟨/pdfmode⟩
⟨/initex⟩
```

Set up the driver for direct PDF output to set the PDF origin equal to TeX's standard origin. The other settings make use of PDF 1.5, which is standard in TeX Live 2011 and should be a reasonable baseline for the future.

```
⟨*initex⟩
⟨*pdfmode⟩
\pdftex_pdfhorigin:D         = 1 true in \scan_stop:
\pdftex_pdfvorigin:D         = 1 true in \scan_stop:
\pdftex_pdfdecimaldigits:D   = 3         \scan_stop:
\pdftex_pdfpkresolution:D    = 600       \scan_stop:
\pdftex_pdfminorversion:D    = 5         \scan_stop:
\pdftex_pdfcompresslevel:D   = 9         \scan_stop:
\pdftex_pdfobjcompresslevel:D = 2        \scan_stop:
⟨/pdfmode⟩
⟨/initex⟩
```

37.2 Driver utility functions

`\__driver_state_save:`
`\__driver_state_restore:`

All of the drivers have a stack for saving the graphic state. These have slightly different interfaces. For both `dvips` and `(x)dvipdfmx` this is done using an appropriate special. Note that here and later, the `dvipdfmx` documentation does not cover the `literal` key word but that this appears to behave in the same way as pdfTeX's `\pdfliteral` (making life easier all-round).

```
⟨*!pdfmode⟩
\cs_new_protected_nopar:Npn \__driver_state_save:
⟨*dvips⟩
  { \tex_special:D { ps:gsave } }
⟨/dvips⟩
⟨*dvipdfmx | xdvipdfmx⟩
  { \tex_special:D { pdf:literal~q } }
⟨/dvipdfmx | xdvipdfmx⟩
\cs_new_protected_nopar:Npn \__driver_state_restore:
⟨*dvips⟩
```

```
  { \tex_special:D { ps:grestore } }
⟨/dvips⟩
⟨*dvipdfmx | xdvipdfmx⟩
  { \tex_special:D { pdf:literal~Q } }
⟨/dvipdfmx | xdvipdfmx⟩
⟨/!pdfmode⟩
```

For direct PDF output there is also a need to worry about the version of pdfTEX in use: the `\pdfsave` primitive was only introduced in version 1.40.0.

```
⟨*pdfmode⟩
\cs_if_exist:NTF \pdftex_pdfsave:D
  {
    \cs_new_eq:NN \__driver_state_save:    \pdftex_pdfsave:D
    \cs_new_eq:NN \__driver_state_restore: \pdftex_pdfrestore:D
  }
  {
    \cs_new_protected_nopar:Npn \__driver_state_save:
      { \pdftex_pdfliteral:D { q } }
    \cs_new_protected_nopar:Npn \__driver_state_restore:
      { \pdftex_pdfliteral:D { Q } }
  }
⟨/pdfmode⟩
```

(*End definition for* `\__driver_state_save:` *and* `\__driver_state_restore:`. *These functions are documented on page* **??**.)

`\__driver_literal:n` The driver code needs to pass on a lot of "raw" information to the underlying binary. The exact command is driver-dependent but the concept is general enough to use a single function. However, it is important to remember this is a convenient shortcut: the arguments will be driver-specific. Note that these functions set the transformation matrix to the current position: contrast with `\@_literal_direct:n`.

```
\cs_new_protected:Npn \__driver_literal:n #1
⟨*dvipdfmx | xdvipdfmx⟩
  { \tex_special:D { pdf:literal~ #1 } }
⟨/dvipdfmx | xdvipdfmx⟩
```

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.

```
⟨*dvips⟩
  {
    \tex_special:D
      {
        ps:
          currentpoint~
          currentpoint~translate~
          #1 ~
          neg~exch~neg~exch~translate
      }
  }
```

```
⟨/dvips⟩
⟨*pdfmode⟩
  { \pdftex_pdfliteral:D {#1} }
⟨/pdfmode⟩
```

(*End definition for* `\__driver_literal:n`.)

`\__driver_literal_direct:n` Even "lower level" than `\@_literal:n`, these commands do not set the transformation matrix but simply dump the driver code directly into the output. In the `(x)dvipdfmx` case this two-part keyword is documented (*cf.* `literal` alone).

```
\cs_new_protected:Npn \__driver_literal_direct:n #1
⟨*dvipdfmx | xdvipdfmx⟩
  { \tex_special:D { pdf:literal~direct~ #1 } }
⟨/dvipdfmx | xdvipdfmx⟩
⟨*dvips⟩
  { \tex_special:D { ps:: #1 } }
⟨/dvips⟩
⟨*pdfmode⟩
  { \pdftex_pdfliteral:D direct {#1} }
⟨/pdfmode⟩
```

(*End definition for* `\__driver_literal_direct:n`.)

`\__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`.

```
⟨*dvips⟩
\cs_new:Npn \__driver_absolute_lengths:n #1
  {
     /savedmatrix~matrix~currentmatrix~def~
     Resolution~72~div~VResolution~72~div~scale~
     DVImag~dup~scale~
     #1 ~
     savedmatrix~setmatrix
  }
⟨/dvips⟩
```

(*End definition for* `\__driver_absolute_lengths:n`.)

`\__driver_matrix:n` Here the appropriate function is set up to insert an affine matrix into the PDF. With a new enough pdfTEX (version 1.40.0 or later) there is a primitive for this, which only needs the rotation/scaling/skew part. With an older pdfTEX or with `(x)dvipdfmx` the matrix also has to include a translation part: that is always zero and so is built in here.

```
⟨*pdfmode⟩
\cs_if_exist:NTF \pdftex_pdfsetmatrix:D
  {
    \cs_new_protected:Npn \__driver_matrix:n #1
      { \pdftex_pdfsetmatrix:D {#1} }
  }
  {
    \cs_new_protected:Npn \__driver_matrix:n #1
```

```
      { \__driver_literal:n { #1 \c_space_tl 0~0~cm } }
  }
⟨/pdfmode⟩
⟨*dvipdfmx | xdvipdfmx⟩
\cs_new_protected:Npn \__driver_matrix:n #1
  { \__driver_literal:n { #1 \c_space_tl 0~0~cm } }
⟨/dvipdfmx | xdvipdfmx⟩
```

(*End definition for* `\__driver_matrix:n`.)

37.3 Box clipping

`\__driver_box_use_clip:N`

The overall logic to clipping a box is the same in all cases. 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 three cases.

```
\cs_new_protected:Npn \__driver_box_use_clip:N #1
  {
    \__driver_state_save:
⟨*dvips⟩
    \__driver_literal:n
      {
        \__driver_absolute_lengths:n
          {
            0~
            \__dim_strip_bp:n { \box_dp:N #1 } ~
            \__dim_strip_bp:n { \box_wd:N #1 } ~
            \__dim_strip_bp:n { - \box_ht:N #1 - \box_dp:N #1 } ~
            rectclip
          }
      }
⟨/dvips⟩
⟨*dvipdfmx | pdfmode | xdvipdfmx⟩
    \__driver_literal:n
      {
        0~
        \__dim_strip_bp:n { - \box_dp:N #1 } ~
        \__dim_strip_bp:n { \box_wd:N #1 } ~
        \__dim_strip_bp:n { \box_ht:N #1 + \box_dp:N #1 } ~
        re~W~n
      }
⟨/dvipdfmx | pdfmode | xdvipdfmx⟩
```

Insert the material in a box of no width, restore the graphic state and then insert the necessary width.

```
    \hbox_overlap_right:n { \box_use:N #1 }
    \__driver_state_restore:
    \skip_horizontal:n { \box_wd:N #1 }
```

```
  }
```

(*End definition for* `\__driver_box_use_clip:N`. *This function is documented on page* 205.)

37.4 Box rotation and scaling

`\__driver_box_rotate_begin:`
`\__driver_box_rotate_end:`

The driver for **dvips** works with a simple rotation angle. In PDF mode, an affine matrix is used instead. The transformation for `(x)dvipdfmx` can be done either way: the affine approach is chosen here as where possible we pick the PDF-style route.

In both cases, some rounding code is included to limit the floating point values to five decimal places. There is no point using any more as TeX's dimensions are of that precision, and the extra figures will simply bloat the PDF and make values harder to trace. In the case where the sine and cosine are used, 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.

```
\cs_new_protected_nopar:Npn \__driver_box_rotate_begin:
  {
    \__driver_state_save:
⟨*dvipdfmx | pdfmode | xdvipdfmx⟩
    \box_set_wd:Nn \l__box_internal_box \c_zero_dim
    \fp_set:Nn \l__box_cos_fp { round ( \l__box_cos_fp , 5 ) }
    \fp_compare:nNnT \l__box_cos_fp = \c_zero_fp
      { \fp_zero:N \l__box_cos_fp }
    \fp_set:Nn \l__box_sin_fp { round ( \l__box_sin_fp , 5 ) }
    \__driver_matrix:n
      {
        \fp_use:N \l__box_cos_fp \c_space_tl
        \fp_compare:nNnTF \l__box_sin_fp = \c_zero_fp
          { 0~0 }
          {
            \fp_use:N \l__box_sin_fp
            \c_space_tl
            \fp_eval:n { -\l__box_sin_fp }
          }
        \c_space_tl
        \fp_use:N \l__box_cos_fp
      }
⟨/dvipdfmx | pdfmode | xdvipdfmx⟩
⟨*dvips⟩
    \fp_set:Nn \l__box_angle_fp { round ( \l__box_angle_fp , 5 ) }
    \__driver_literal:n
      {
        \fp_compare:nNnTF \l__box_angle_fp = \c_zero_fp
          { 0 }
          { \fp_eval:n { - \l__box_angle_fp } }
        \c_space_tl
        rotate
      }
```

```
⟨/dvips⟩
  }
```

The end of a rotation means tidying up the output grouping.

```
\cs_new_eq:NN \__driver_box_rotate_end: \__driver_state_restore:
```

(*End definition for* `\__driver_box_rotate_begin:` *and* `\__driver_box_rotate_end:`*. These functions are documented on page 206.*)

`\__driver_box_scale_begin:`
`\__driver_box_scale_end:`

Scaling is not dissimilar to rotation, but the calculations are somewhat less complex.

```
\cs_new_protected_nopar:Npn \__driver_box_scale_begin:
  {
    \__driver_state_save:
    \fp_set:Nn \l__box_scale_x_fp { round ( \l__box_scale_x_fp , 5 ) }
    \fp_set:Nn \l__box_scale_y_fp { round ( \l__box_scale_y_fp , 5 ) }
⟨*dvips⟩
    \__driver_literal:n
      {
        \fp_use:N \l__box_scale_x_fp \c_space_tl
        \fp_use:N \l__box_scale_y_fp \c_space_tl
        scale
      }
⟨/dvips⟩
⟨*dvipdfmx | pdfmode | xdvipdfmx⟩
    \__driver_matrix:n
      {
        \fp_use:N \l__box_scale_x_fp \c_space_tl
        0~0~
        \fp_use:N \l__box_scale_y_fp
      }
⟨/dvipdfmx | pdfmode | xdvipdfmx⟩
  }
\cs_new_eq:NN \__driver_box_scale_end: \__driver_state_restore:
```

(*End definition for* `\__driver_box_scale_begin:` *and* `\__driver_box_scale_end:`*. These functions are documented on page 206.*)

37.5 Color support

`\l__driver_current_color_tl`

The current color is needed by all of the engines, but the way this is stored varies.

```
\tl_new:N \l__driver_current_color_tl
⟨*dvipdfmx | xdvipdfmx⟩
\tl_set:Nn \l__driver_current_color_tl { gray~0 }
⟨/dvipdfmx | xdvipdfmx⟩
⟨*dvips⟩
\tl_set:Nn \l__driver_current_color_tl { Black }
⟨/dvips⟩
⟨*pdfmode⟩
\tl_set:Nn \l__driver_current_color_tl { 0~g~0~G }
⟨/pdfmode⟩
```

(*End definition for* `\l__driver_current_color_tl`*. This variable is documented on page* **??**.)

\l__driver_color_stack_int

pdfTEX (version 1.40.0 or later) and LuaTEX have multiple stacks available, and the color stack therefore needs a number when in PDF mode.

```
⟨*pdfmode⟩
\int_new:N \l__driver_color_stack_int
⟨/pdfmode⟩
```

(*End definition for* \l__driver_color_stack_int. *This variable is documented on page* ??.)

__driver_color_ensure_current:
__driver_color_reset:

Setting the current color depends on the nature of the color stack available. In all cases there is a need to reset the color after the current group.

```
⟨*dvipdfmx | dvips | xdvipdfmx⟩
\cs_new_protected_nopar:Npn \__driver_color_ensure_current:
  {
    \tex_special:D { color~push~\l__driver_current_color_tl }
    \group_insert_after:N \__driver_color_reset:
  }
\cs_new_protected_nopar:Npn \__driver_color_reset:
  { \tex_special:D { color~pop } }
⟨/dvipdfmx | dvips | xdvipdfmx⟩
```

Once again there is a version switch for pdfTEX, as the \pdfcolorstack primitive was introduced in version 1.40.0.

```
⟨*pdfmode⟩
\cs_if_exist:NTF \pdftex_pdfcolorstack:D
  {
    \cs_new_protected_nopar:Npn \__driver_color_ensure_current:
      {
        \pdftex_pdfcolorstack:D \l__driver_color_stack_int push
          { \l__driver_current_color_tl }
        \group_insert_after:N \__driver_color_reset:
      }
    \cs_new_protected_nopar:Npn \__driver_color_reset:
      { \pdftex_pdfcolorstack:D \l__driver_color_stack_int pop }
  }
  {
    \cs_new_protected_nopar:Npn \__driver_color_ensure_current:
      {
        \__driver_literal:n { \l__driver_current_color_tl }
        \group_insert_after:N \__driver_color_reset:
      }
    \cs_new_protected_nopar:Npn \__driver_color_reset:
      { \__driver_literal:n { \l__driver_current_color_tl } }
  }
⟨/pdfmode⟩
```

(*End definition for* __driver_color_ensure_current:. *This function is documented on page 206.*)

```
⟨/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.

Symbols

Numbers

A

C

E

F

G

H

M

N

Q

R

S

T

W

X

Y